Lists and Trees

List Ranking, Euler Tour, Tree Traversal,
Expression Evaluation, Range Minima, LCA,

Linked Lists

*[Recap from 2120]

Linked List Basics – Warm-up☆

- There are two basic mechanisms to work with lists (collection of items of similar items): (1) Arrays (2) Linked Lists
- **Arrays:** Advantages: once an array is declared and set up, the [ ] operators makes random access extremely fast on hardware using fast address arithmetic. Disadvantages: The size needs be declared (can be done only once) – no dynamic allocation of space – wastage of space in most cases [programming efforts using realloc() in the heap area can minimize – inconvenient, complex]. Insertion/deletion of elements is time consuming [existing elements must be moved around]
- **Linked lists**: It can dynamically allocate/deallocate storage with minimal effort and insertion/deletion of elements is also fast – there are many other potential advantages to create new types of structures. Random access is slow, no [ ] operator.
- **Lesson**: Choose your data structure to suit your application.
- Read here for linked list basics; Read here for a refresher on C. Remember: pointers should not be used without first initializing them [both C and C++]
- We will write a C program to build a linked list along with functions for insertion, deletion, search, create header etc. [experiment with it, be comfortable.]

Simple implementation in C☆

```
typedef struct{
     int info;
      struct node *link;
}node;
void create (node **root){
   *root =NULL; }
void insert (node **lp, int key){
node *current, *prev, *new;

current = *lp; prev = NULL; //start at list head
while (current != NULL && current->info < key)
  {prev =current;
    current = current->link;}
    if (current != NULL &&  current->info == key)
     printf ("The key already exists; Try again\n");
  else {new = (node *)  malloc(sizeof(node));
      new->info = key;
      new->link = current;
      if (prev == NULL) *lp =new
      else prev->link = new;  }
                                }
```

- **Assume**: we maintain a sorted list of integers; insertions/deletions must keep the list always sorted.
- Each element in a linked list is usually called a node that has two components: the element and a pointer that points to the next element in the list. Use a struct.
- Pass parameters by reference if they are to be changed by a function.
- Insert will insert a new node with a key in the list in sorted order iff the key is not already present in the list. Need two pointers to scan the list – current and previous.
- Once the correct position is found with no error), create a new node and adjust the pointers accordingly (depending on if the newly inserted node is the first element of the list or not).

Code in C (contd.)☆

```
void delete (node **lp, int key){

node *current, *prev;
current = *lp; prev = NULL;
 while (current != NULL && current->info < key)
   {prev =current;
   current = current->link;}

   if (current->info == key)
      {if (prev == NULL) {*lp = current->link;}
      else {prev->link = current->link; free ((void *)
                current);};}
      else printf ("Key is not in list \n");
}

void printlist (node **lp){
  node *current =*lp;
  while (current != NULL)
  {printf ("%d\n", current->info);
     current = current->link;}
}
```

- Use the functions to process a list: node *root; insert (&root, 4); insert (&root, 5); insert (&root, 10); insert (&root, 15); insert (&root, 8); printlist(&root); delete(&root, 10): and so on.
- **Exercise**:
  - Design functions for "multi-insert", "multi-delete". Design a function to empty an existing list (you need to garbage collect); functions like is_empty(list).
  - Design a function that inserts new elements at the end of a list (not to be sorted). Think of many others.
  - Design a function to compute the length of a list.
  - Next, consider a circular list, a stack, and a queue and a deque.
  - Think of doubly linked or quadruply linked lists.
  - Also, consider having a "head" node to make the operations efficient

Stacks☆

- In a **stack**, the element deleted from the set is the one most recently inserted: the stack implements a **last-in, first-out, or LIFO**, policy. Similarly, in a **queue**, the element deleted is always the one that has been in the set for the longest time: the queue implements a **first-in, first-out, or FIFO**, policy.
- The INSERT operation on a stack is often called **PUSH**, and the DELETE operation, which does not take an element argument, is often called **POP**. We can implement a stack of at most n elements with an array S[1… n]. The array has an attribute S.top that indexes the most recently inserted element; if S.top = 0, the stack is empty. We test to see whether the stack is empty by query operation STACK-EMPTY; if we attempt to pop an empty stack, we say the stack **underflows**, which is normally an error; if S.top exceeds n, the stack **overflows**.
  - IS_EMPTY(S): {**if** S.top = 0 **then** return True **else** return False}
  - PUSH(S, x): {if S.top = n **then** exit (overflow) **else** {S.top++; S[S.top] = x}}
  - POP(S): {**if** IS_EMPTY(S) **then** exit (underflow) **else** {S.top--; return S[S.top + 1}}
- Each of the 3 operations takes O(1) [constant] time. One can think of many other such functions, like SIZE, EMPTY(), FIND(k), etc.
- Note that the definition of the ADT does not just list the operations. It specifies their meaning, including their effects on the value of the stack. It also specifies the possible errors that can occur when operations are applied. Errors are a particularly difficult thing to deal with in an abstract way.
- Example Applications of Stacks: Page-visited history in a Web browser, Undo sequence in a text editor, Chain of method calls in a language supporting recursion, Auxiliary data structure for many algorithms, Component of other data structures

Simple C++ Stack Implementation using Array

Typically in C++, a class is defined in two files, a header (or ".h") file and an implementation (or "cc") file:

```
// The header file, intstack.h
class IntStack {
  public:
    IntStack(); // A constructor for this class.
    void push(int x); // The operations of the Stack
ADT.
     int pop();
     void makeEmpty();
     bool isEmpty();
  private:
  int data[100]; // Contains the items on the stack.
  int top; // Number of items on the stack.
};
```

Note: The data for the stack is stored in the private part of the class; this ensures that if the data representation is changed, code that uses the class will not have to be modified but has to be recompiled. (why?)

```
// The file intstack.cc
#include "intstack.h"
IntStack::IntStack() {
top = 0; // A stack is empty when it is first created. }
void IntStack::push(int x) {
  if (top == 100) // ERROR; throw an exception.
   cerr << "Attempt to push onto a full stack";
   data[top] = x;
    top++;   }
int IntStack::pop() {
  if (top == 0) // ERROR; throw an exception
   cerr <<  "Attempt to pop from an empty stack";
   top--;
    return data[top];  }
void IntStack::makeEmpty() {
   top = 0;   }
bool IntStack::isEmpty() {
   return (top == 0);   }
```

Exercise: We cannot push more than 100 integer. Change the "push" operation so that when the stack is about to overflow, allocate a stack of twice the size, copy the original elements and deallocate the old stack (use a destructor ~IntStack); start with writing the pseudocode.

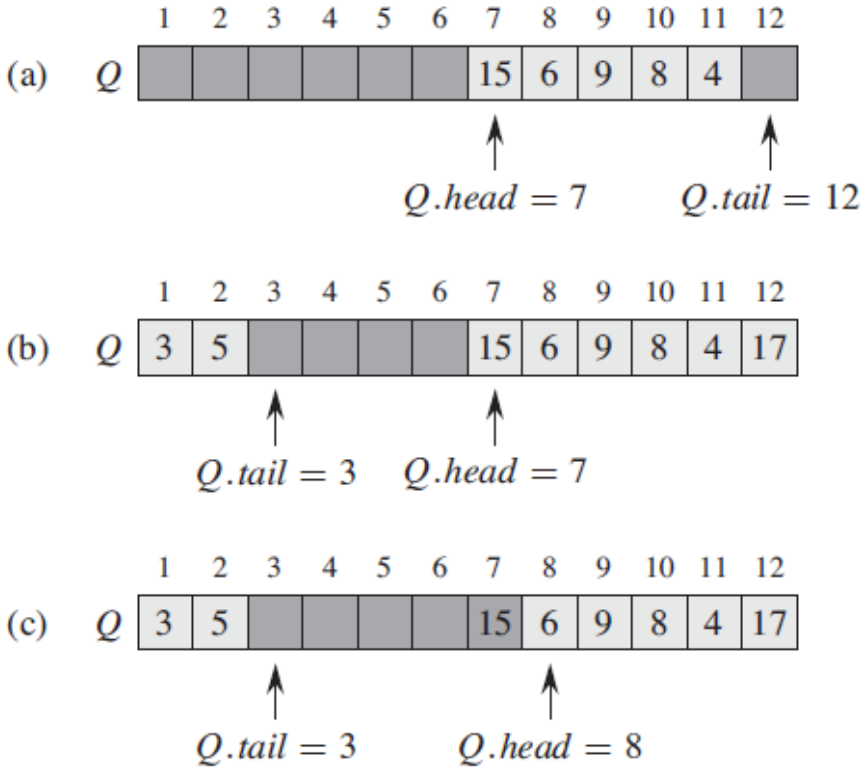
Two Simple Applications of Stacks☆

- Evaluate arithmetic expressions: Say, x = 4 – 3 * 2/ (7 – 4) + 8; Remember the precedence rules of the operators; within the same precedence class operators are executed in a left to right fashion. Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations. Write the complete code.
- Given an array X, the span S[i] of X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] and such that X[j] ≤ X[i]. Spans have applications to financial analysis, e.g., stock at 52-week high. Say, X = {6, 3, 4 , 5, 2}, then S = {1, 1, 2, 3, 1}; Given X, compute S. We may do this in detail later.

Queues☆

- Queues are ADTs that provide access at both ends, unlike the stacks. We call the INSERT operation on a queue **ENQUEUE**, and we call the DELETE operation **DEQUEUE**. The queue has a **head** and a **tail** (alternatively, front and rear). When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.
- One way to implement a queue of at most n – 1 elements using an array Q[1… n]. The queue has an attribute Q.head that indexes, or points to, its head. The attribute Q.tail indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations Q.head, Q.head + 1, … Q.tail − 1, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order. When Q.head = Q.tail = 1, the queue is empty. Initially, we have Q.head = Q.tail = 1. When Q.head = Q.tail+1, the queue is full. Assume n = Q.length
- Enque(Q,x): {if Q.head = Q.tail+1 then exit (overflow) else {Q[Q.tail] = x; if Q.tail = Q.length then Q.tail=1 else Q.tail++}}
- Dequeue(Q): {if Q.head = Q.tail = 1 then exit (underflow) else {x = Q[Q.head]; if Q.head = Q.length then Q.head = 1 else Q.head++}}
- Each operation takes O(1) time. Note that we can not use the last element of the array; why? Is there a way to eliminate that minor problem? Think!

Example and Exercise☆



Applications: Buffering videos in packets, waiting lists, access to shared resources, multiprogramming (e.g., round robin), auxiliary data structure for many algorithms, components of other data structures, etc.

- Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a **deque** (double-ended queue) allows insertion and deletion at both ends. Write four O(1) time procedures to insert elements into and delete elements from both ends of a deque implemented by an array (or a linked list).
- How do we implement a circular queue? Can we adjust our functions (classes) to handle the overflow problem by increasing (doubling) the size of the queue?

Linked Lists☆

- Stacks, Queues and Deques belong to a general class of data structure, called List. We have briefly refreshed how lists can be implemented using arrays (contiguous memory space) – the advantage is that most key operations are O(1) due to possible indexing in arrays; the disadvantage is we cannot do insertion or deletions at arbitrary positions in O(1) time (re-compaction will take O(n) worst time) – not very effective in applications that need frequent insertions and deletions. Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme in which each node contains a link to the next node of the list.
  - Linked allocation (both **singly linked** or **doubly linked**) takes up additional memory space for the links.
  - It is not necessary to know the addresses of all current keys – adverse effect is searching for a specific key is of O(n) complexity in the worst case – once the position is located, insertion/deletion takes O(1) time.
  - The linked scheme makes it easier to join two lists together, or to break one apart into two that will grow independently [Not generally possible when using arrays].
  - The linked scheme lends itself immediately to more intricate structures than simple linear lists. We can have a variable number of variable-size lists; any node of the list may be a starting point for another list; the nodes may simultaneously be linked together in several orders corresponding to different lists; and so on.
  - Read here for linked list basics; Read here for a refresher on C. **Remember**: pointers should not be used without first initializing them [both C and C++]

Code in C (contd.)

```
void delete (node **lp, int key){

node *current, *prev;
current = *lp; prev = NULL;
 while (current != NULL && current->info < key)
   {prev =current;
   current = current->link;}

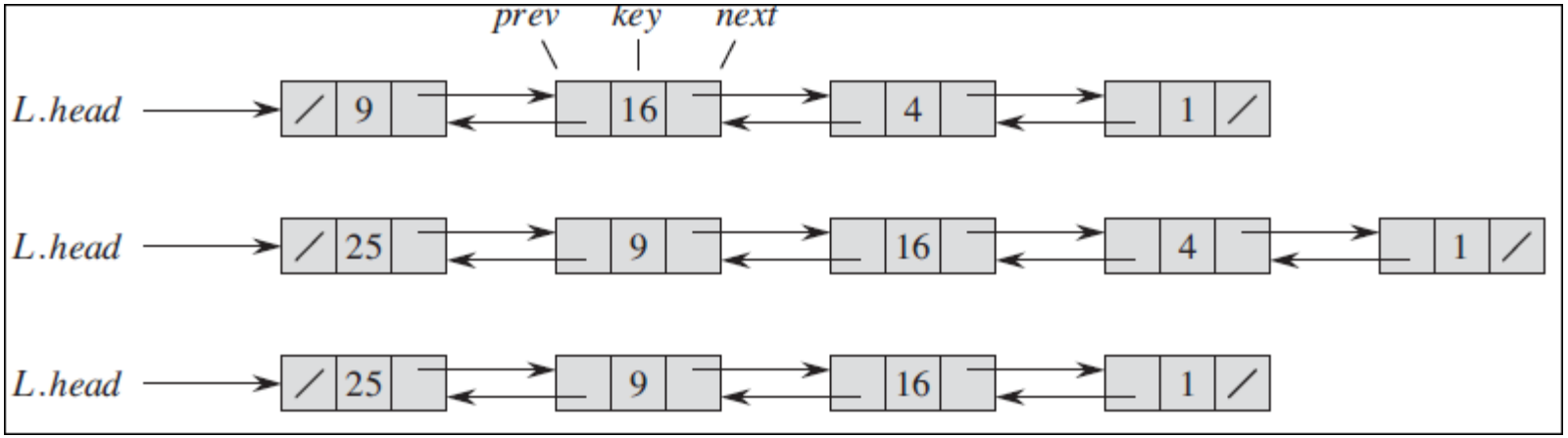
   if (current->info == key)
      {if (prev == NULL) {*lp = current->link;}
      else {prev->link = current->link; free ((void
*)                current);};}
      else printf ("Key is not in list \n");
}

void printlist (node **lp){
  node *current =*lp;
  while (current != NULL)
  {printf ("%d\n", current->info);
     current = current->link;}
                                  }
```

- Use the functions to process a list: node *root; create (&root); insert (&root, 4); insert (&root, 5); insert (&root, 10); insert (&root, 15); insert (&root, 8); printlist(&root); delete(&root, 10): and so on.
- **Exercise**:
  - Design functions for "multi-insert", "multi-delete". Design a function to empty an existing list (you need to garbage collect); functions like is_empty(list).
  - Design a function that inserts new elements at the end of a list (not to be sorted). Think of many others.
  - Design a function to compute the length of a list.
  - Next, consider a circular list, a stack, and a queue and a deque.
  - Think of doubly linked or quadruply linked lists.
  - Also, consider having a "head" node to make the operations efficient

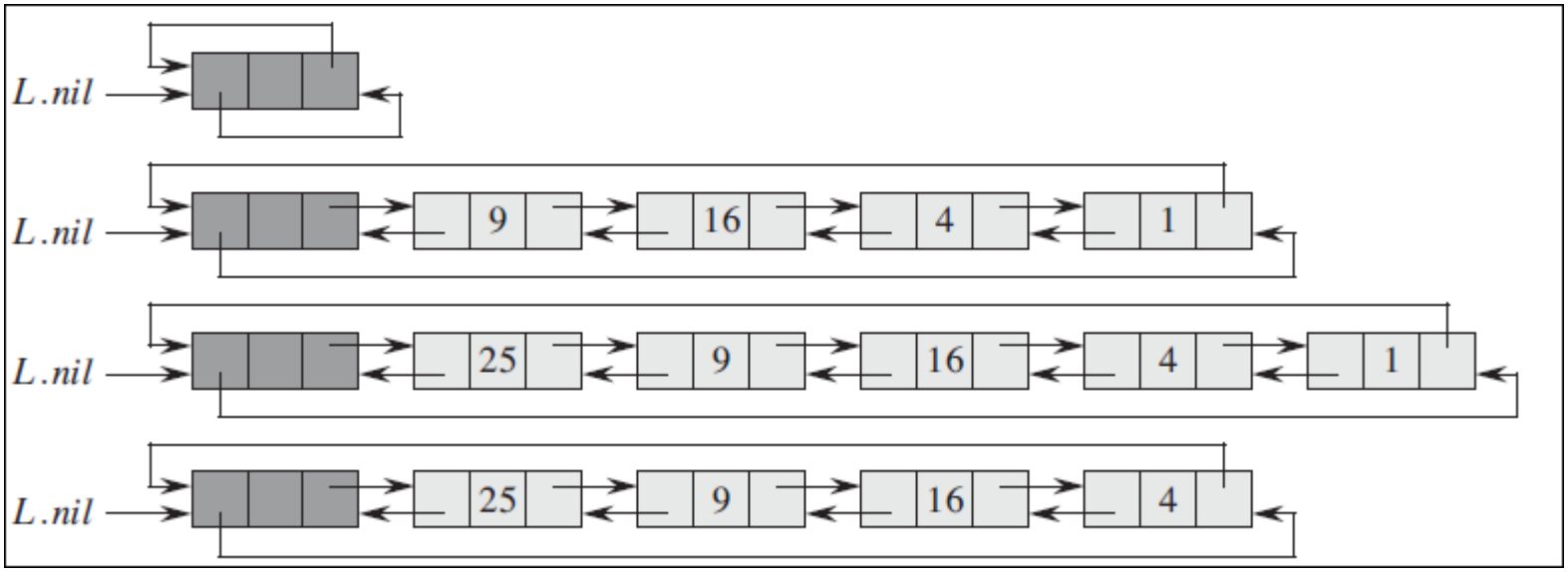
Doubly Linked Lists ☆

- Use two links prev and next to facilitate traversing the list from two ends



Circular Linked Lists☆

- Using headers and/or making them circular



- One needs to adjust the code to accommodate the header and circularity
- Things to note:
  - lists are to be always sorted or not. Insert and delete operations may be defined in different ways.
  - Headers can contain information (e.g., size of the list) about the list and or other things.
  - Implement stacks, queues, deques using linked lists.

Further Exercise Problems, Singly Linked lists☆

- Singly Linked Lists (with or without a special header node)
- **Remove Duplicates** if any exists
- **FrontBackSplit():** Given a list, split it into two sublists — one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So FrontBackSplit() on the list {2, 3, 5, 7, 11} should yield the two lists {2, 3, 5} and {7, 11}. Easy if size is known from header (if not, compute). Or run two pointers – one slow and the other twice as fast – when fast pointer reaches the end, the slow pointer will be about halfway.
- **MoveNode()**: This is a variant on Push(). Instead of creating a new node and pushing it onto the given list, MoveNode() takes two lists, removes the front node from the second list and pushes it onto the front of the first.
- **void AlternatingSplit**(node *source, node **first, node **second): takes one list and divides up its nodes to make two smaller lists. The sublists should be made from alternating elements in the original list.
- **Reverse a list**: Ideally, Reverse() should only need to make one pass of the list. The iterative solution is moderately complex. Recursive Reverse is quite tricky to make it scan the list only once.
- Merge two sorted lists to produce a bigger sorted list: a bit tricky if you are asked to do it in place.

List Ranking

What is List Ranking?

- One of the most 'elementary' list-processing tasks is to rank the nodes from either end of the list – the list-ranking problem.
- We consider a list of n nodes whose order is specified by an array S such that S(i) = j when j is the node following node i in L, $1 \leq i \leq n$ [S(i) = 0 if i is the last node of the list]. We want to determine the distance of each node from the end of the list.
- Given a linked list L with n nodes, we would like to compute an array R such that R(i) is equal to the distance of node i from the **end** of L. Initially, we set R(i) = 1 for all nodes i, except for the last node, whose R value is set to 0.
- Sequential algorithm is trivial – reverse the direction in one scan and then traverse again while computing distance – T*(n) = O(n), W*(n) = O(n) – linear run time, linear work (can use either arrays or linked lists – there is a difference when we want to parallelize).
- How fast can we do in parallel? Where is the parallelism?

Use Pointer Jumping

```
begin
  1. for 1 ≤ i ≤ n pardo
        if (S(i) ≠ 0) then set R(i): = 1
                      else set R(i): = 0
  2. for 1 ≤ i ≤ n pardo
        Set Q(i): = S(i)
        while (Q(i) ≠ 0 and Q(Q(i)) ≠ 0) do
              Set R(i): = R(i) + R(Q(i))
              Set Q(i): = Q(Q(i))
end
```

Note: Almost identical to finding roots (for each node) in a forest. R is the output array giving the distance; using Q is not essential; Parallel run time is O(log n) and total number of operations is O(n log n) – not work-optimal! How to do better?

Smart Strategy

- **Shrink** [don't know how yet] the list until only O(n/log n) nodes remain – this is the big question; this shrinking needs be done in O(log n) time using linear operations.
- **Use pointer jumping on the short list** of size O(n/log n) $\Rightarrow$ this can be done in O(log(n/log n)) = O(log n) time using O((n/log n)*log(n/log n)) = O(n) operations.
- **Restore the original list** and rank the nodes removed in Step 1 $\Rightarrow$ essentially reverse the step 1 if we do things correctly.
- The method for shrinking the list L consists of removing a selected set of nodes from L and updating the intermediate R values of the remaining nodes. The key to a fast parallel implementation lies in using an **independent** set of nodes to be removed.
- We need some concepts and techniques – let us see what and how.

Concept of an Independent set

A set of nodes I (in a list L) is called an independent set if $i \in I \Rightarrow S(i) \notin I$.

1. We can remove nodes from I by adjusting the successor pointer of the predecessor node.
2. Since I is independent, this operation can be done in parallel [does not affect neighbor nodes]
3. Information about the removed nodes must be stored appropriately since they need be put back to get the desired solution.
4. The method for shrinking the list L consists of removing a selected set of nodes from L and updating the intermediate R values of the remaining nodes.
5. The information concerning the removed nodes should be stored somewhere so that later the original list can be restored and the nodes in I can be ranked properly. A separate array is needed because our list contraction procedure will compact the remaining elements into consecutive memory locations. Without loss of generality, we assume that the predecessor array P is available. Otherwise, it can be set up in O(1) time using O(n) operations.

Remove nodes from an Independent Set

Input: (1) Arrays S and P of length n representing, respectively, the successor and the predecessor relations of a linked list; (2) an independent set I of nodes such that P(i), S(i) ≠ 0; (3) a given value R(i) for each node i.

Output: The list obtained after removal of all the nodes in I with the updated R values.

begin

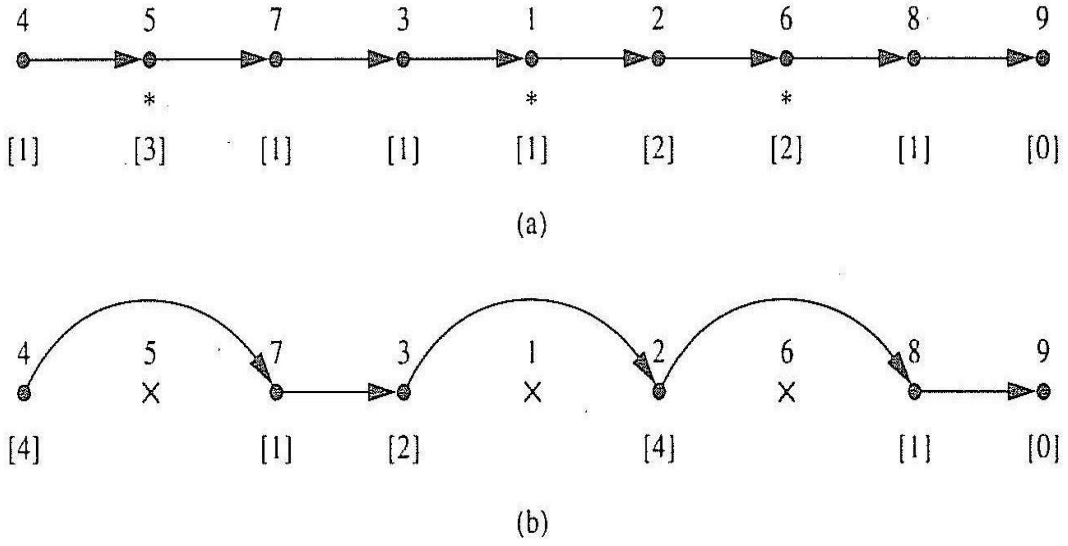
1. Assign consecutive serial numbers N(i) to the elements of I, where $1 \le N(i) \le |I| = n'$.
   [How to do this?]
   for $1 \le i \le n$ pardo
   if $i \notin I$ then N(i) = 0 else N(i) =1
   Do a prefix sum on N[] in parallel
2. **for all** $i \in I$ **pardo**
   Set U(N(i)) : = (i, S(i), R(i))·
   Set R(P(i)) : = R(P(i) + R(i))
   Set S(P(i)) : = S(i)
   Set P(S(i)): = P(i)

end

Note: Step 1 takes O(log n) time using O(n) operations by a prefix-sums computation on the nodes of L such that a weight of 1 is assigned to each node in I, and a weight of 0 is assigned to each of the remaining nodes [N(i)]. Step 2 can be executed in O(1) time, using O(n) operations.

Example of List Contraction

R(i) is equal to the distance of node i from the end of the list



(a) is the initial list. The R values are given in brackets below the nodes, and the selected nodes to be removed are labeled with star *.

(b) is the resulting list after contraction.

- An independent set of nodes consists of {1, 5, 6}. Step 1 assigns to the nodes {1, 5, 6} the serial numbers N(1) = 1, N(5) = 2, and N(6) = 3. Executing the loop of step 2 for node 6, we obtain U(N(6)) = U(3) = (6, S(6), R(6)) = (6,8,2), R(P(6)) = R(2) = 4, S(P(6)) = S(2) = 8, and P(S(6)) = P(8) = P(6) = 2.
- We proceed similarly for the nodes 5 and 1. The new links and the new R values of the contracted list are shown in (b).
- Suppose we now have the ranks of all the nodes on the short list considered with the updated R values, say using pointer jumping. Now, we need to reinsert the removed nodes. *We need to have a way to know an independent set*.

Reinsert the removed nodes

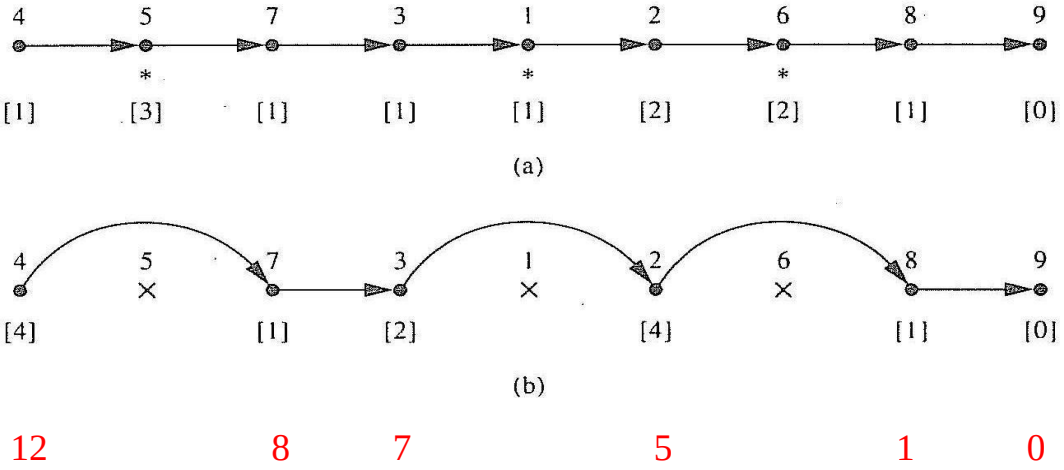
for all i ∈ I **pardo**

U(N(i)) = (i, S(i), R(i))

R(P(i)) = R(P(i)) + R(i)

S(P(i)) = S(i)

P(S(i)) = P(i)



Suppose we now have the ranks of all the nodes on the short list considered with the updated R values (shown in red). Since U(3) = (6, 8, 2), we conclude that node 8 is the initial successor of node 6, and initial weight at node 6 was 2 – thus, after re-insertion of node 6, we have R(6) = R(8) + 2 = 3. Moreover, we can reinsert this node by setting P(6) = P(8) = 2, S(P(6)) = S(2) = 6, P(8) = 6. We can reinsert the nodes 1 and 5 similarly.

Note

This reinsertion step requires O(1) time using O(n) operations. If we execute this on a set of p processors, where $p \leq |I|$, the n' elements of I will be divided almost evenly into p blocks, each of which will be handled separately by a processor. Each processor can push on a private stack the information concerning the nodes it removes. Restoring the list will be accomplished by popping each such stack.

How to get an Independent Set

- k-coloring of the nodes in a list is defined to be a mapping of the nodes into {0, 1, 2, …, k-1} such that no two adjacent nodes have the same color.
- A node is called the local minimum w.r.t. a k-color if it has a smaller color than its predecessor and the successor.
- Given a list of n nodes and a k-coloring, the set of local minima is an independence set and is of size $\Omega(n/k)$.
- Why? Consider 2 consecutive minima nodes u and v: u and v cannot be adjacent; colors of nodes between u and v form an increasing sequence followed by a decreasing sequence $\Rightarrow$ maximum number of nodes between u and v is 2k-3.
- It takes O(1) time if a node is local minimum by comparing its successor and predecessor and can be done in parallel for all nodes $\Rightarrow$ the set can be computed in O(1) time using O(n) operations.

Important Observations

- We can obtain a large independent set by using the optimal algorithm to 3-color the vertices of a cycle.
- The corresponding independent set is of size greater than or equal to cn, for some constant $0 < c < 1$ (more precisely, we have $c = 1/5$ if 3-coloring is used). Explain why? [consider small lists and their worst-case scenarios]
- Removal of nodes from this independent set reduces the size of the list by a constant factor; hence, this process can be repeated $\alpha * \lceil \log \log n \rceil$ times to produce a list of size less than or equal to $n/\log n$, for some $\alpha > 0$ [choose $\alpha = 4/5$ for 3-coloring].
- Note: $n \rightarrow (4/5)n \rightarrow (4/5)^2 n \rightarrow (4/5)^3 n \rightarrow \ldots \rightarrow (4/5)^x n = n/\log n$ which gives $(4/5)^x = \log n$, or $x = (5/4)\log \log n$

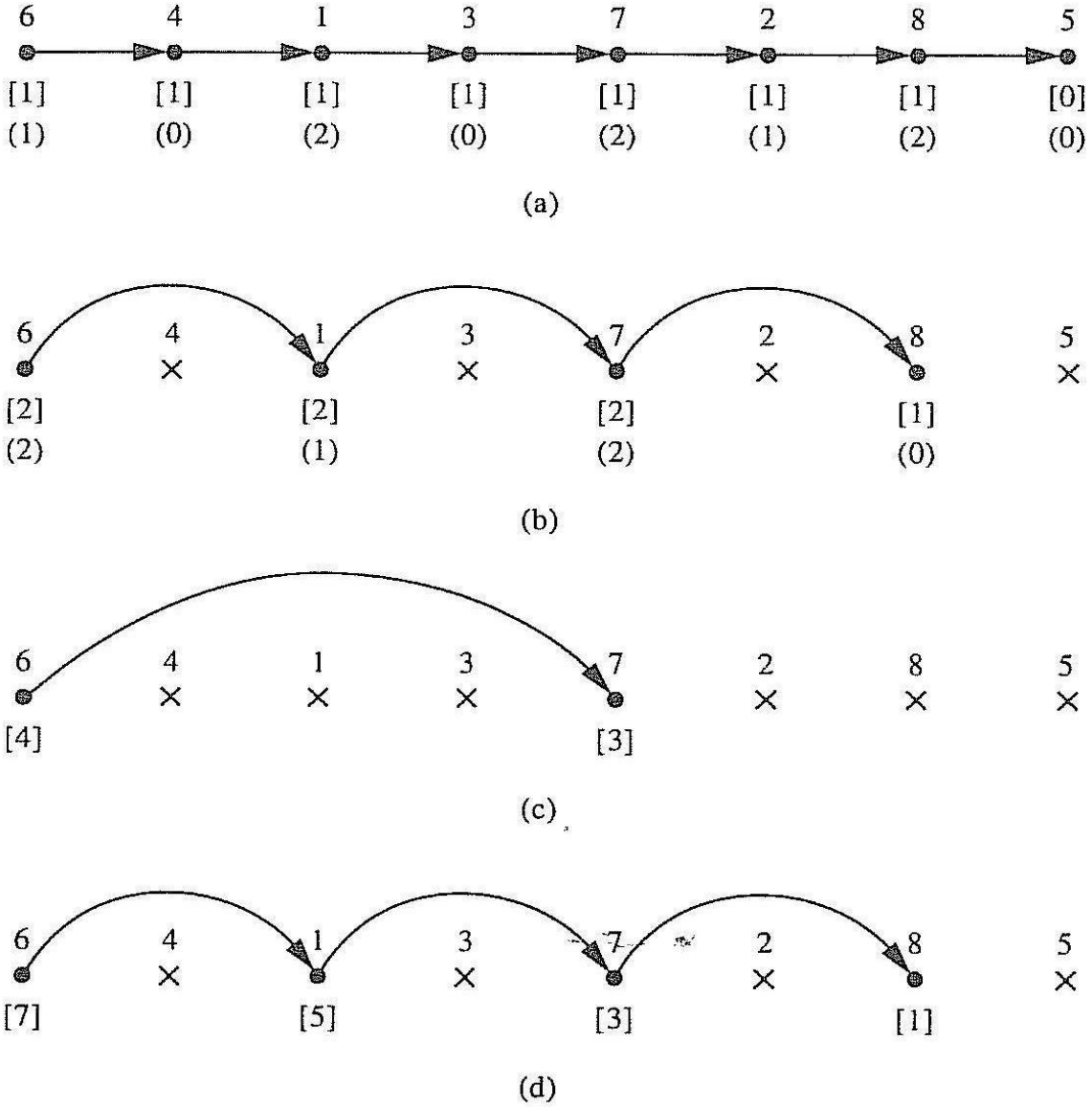
Now we can get the complete algorithm

Complete List Ranking

begin

1. Set n_0: = n, k: = 0
2. while n_k > n/log n do
   - Set k: = k + 1.
   - Color the list with three colors, and identify the set I of local minima.
   - Remove the nodes in I, and store the appropriate information regarding the removed nodes.
   - Let n_k be the size of the remaining list. Compact the list into consecutive memory locations.
3. Apply the pointer jumping technique to the resulting list.
4. Restore the original list and rank all the removed nodes by reversing the process performed in step 2.

end



The restoration takes 2 iterations. After the first iteration (corresponding to the second iteration of the while loop), we obtain the list shown in (d). It is clear that the second iteration will fully restore the original list with correct ranks at all nodes.

Weight of each node is in brackets. During the first iteration, we identify the independent set I = {3, 4, 2, 5} from 3-coloring (example) shown in parentheses. We get new list shown in (b) with adjusted weights. [info of removed nodes are stored with an additional time stamp indicating the iteration number; For example, the information related to node 4 can be stored as U(1, N(4)) = (4,5(4), R(4)) = (4,1, 1), where parameter (1) of U indicates the iteration number and N(4) indicates the serial number assigned to the node being removed. Next, we obtain the list (c). Since $n_z = 2$, we go to step 3 and obtain R(6) = R(6) + R(7) = 7 and R(7) = 3.

Analysis Sketch

- If n_k is the size of the list at the beginning of $(k+1)^{st}$ iteration of Step 2, $|I| \geq n_k/5$, we have $n_{k+1} \leq 4n_k/5$ and $n_k \leq (4/5)^k n$. Thus, number of iterations needed to reduce the size of the list below n/log n is O(log log n).
- Optimal 3-coloring runs in O(log n) time using linear operations; resource needs for step 2.2 and 2.3 are known.
- Step 4 needs labeling each node by 1 or 0 and performing a prefix sum which takes O(log n) time using O(n) operations.. Thus, total time taken by step 2 is O(log n log log n) using $O(\Sigma n_k) = O(\Sigma(4/5)^k n = O(n)$ operations.
- Step 3 can be executed in O(log n) time using O(n) operations; Step 4 is obvious.
- Thus, the resulting algorithm is work-optimal and runs in O(log n log log n) time.
- It is possible to get a work-optimal algorithm that takes O(log n) time – relatively complicated

k – Coloring of a Ring

- Note that k-coloring is a mapping function c: V $\rightarrow$ {0, 1, 2 ... k-1} such that c(i) $\neq$ c(j) if (i,j) $\in$ E, where the graph is given as G = (V, E); c() is the color array indexed by the node numbers.
- Consider the problem of 3-coloring a directed cycle. Assume that the directed cycle is specified by a successor array S of length n (where n is the number of nodes in the cycle such that S(i) = j whenever (i,j) $\in$ E. The input is the S array, and we are to generate the c-array using elements from {0,1,2}. The sequential algorithm is straightforward – it takes O(n) time and does O(n) work.
- It is trivial to design a sequential algorithm: Scan the array from beginning to end and marking the color of a node either R or B alternately (to satisfy the coloring definition) – the last node will be special if the size of the ring is even, we will need a third color. Thus, an arbitrary ring cannot be colored in less than 3 colors.
- How to parallelize it? Important thing to note is that the nodes are numbered randomly.
- The problem looks like an inherently sequential problem – we need to *break the symmetry*

Basic Coloring

- To start with we set c(i) = i for all i, $1 \le i \le n$. Certainly, this is a valid coloring (each node has a distinct color) and we can do it in one parallel step and doing O(n) work. But this is not 3-coloring. Next, from the c-array we generate another array c1 executing the following code:

  for i = 1 to n **pardo**

  S1: set k = least significant bit position where c(i) and c(S(i)) disagree.

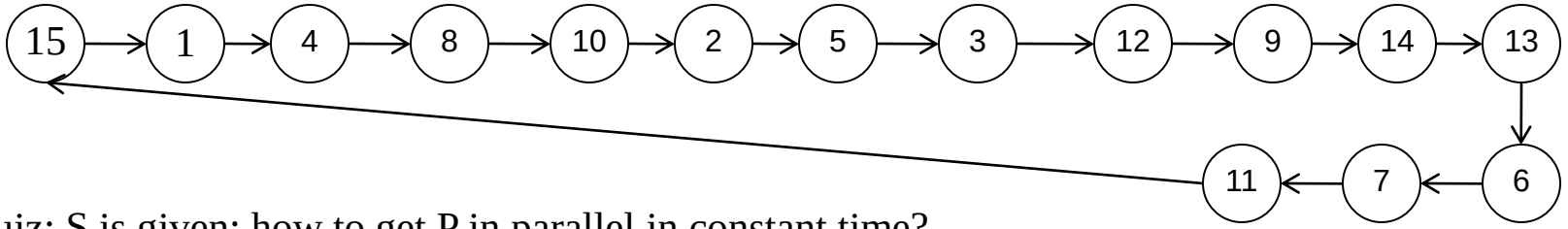
  S2: $c1(i) = 2k + c(i)_k$

- We claim that this new array c1 is also a valid coloring – not a 3-color yet but uses far less colors than the array c does (c uses n colors). We observe
  - Since the color c is valid, the integer k will always exist for each i in step S1. *Why*?
  - To prove the validity of c1, assume otherwise, i.e., there exists c1(i) = c1(j) for some i and j such that S(i) = j. Let $c1(i) = 2k + c(i)_k$ and $c1(j) = 2m + c(j)_m$.
  - Since c1(i) = c1(j) [by assumption], we must have k = m (because both $c(i)_k$ and $c(j)_m$ are either 0 or 1). Hence, $c(i)_k = c(j)_m = c(j)_k$ – contradiction by the definition of the integer k in the algorithm.
  - The above algorithm to generate c1-array from c-array runs in O(1) time and does O(n) work, assuming the step S1 (the job of computing the least significant bit position where two binary numbers disagree) can be done in O(1) time. Assume, for the time being we have such an algorithm (will show it soon).

An Example

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	4	5	12	8	3	7	11	10	14	2	15	9	6	13	1
P	15	10	5	1	2	13	6	4	12	8	7	3	14	9	11

v	15	1	4	8	10	2	5	3	12	9	14	13	6	7	11
c	1111	0001	0100	1000	1010	0010	0101	0011	1100	1001	1110	1101	0110	0111	1011
k	1	0	2	1	3	0	1	0	0	0	0	0	0	2	2
c1	3	1	5	2	7	0	2	1	0	1	0	1	0	5	4



Quiz: S is given; how to get P in parallel in constant time?

Superfast 3-coloring of a cycle

- Consider the previous algorithm of basic coloring. Assume "t" bits are needed to do the initial coloring c. Then coloring c1 will need $t1 = \lceil \log_2 t \rceil + 1$ number of bits. In other words, if coloring c uses q colors, $2^{t-1} < q \leq 2^t$, then coloring c1 uses at most $2^{\lceil \log\ t \rceil + 1}$ colors or O(t) = O(log n) colors. Thus, number of colors decreases exponentially in one parallel step.
- Apply the algorithm repeatedly. The number of colors decreases if $t > \lceil \log t \rceil + 1$ or $t > 3$. When t =3, i.e., coloring c consists of at most 8 colors, coloring c1 contain at most 6 colors [c1(i) = 2k + (either 0 or 1), where $0 \leq k \leq 2$, so $0 \leq c1(i) \leq 5$].
- Let $\log^{(i)} x = \log (\log^{(i-1)} x)$ where $\log^{(1)} x = \log x$ and let $\log^* x = \min \{i, \mid \log^{(i)} x \leq 1\}$. Observe that $\log^* x$ is an extremely slowly increasing function [$\log^* x \leq 5$ for all $x \leq 2^{65536}$].
- Thus, starting with an initial coloring, c(i) = i for all i, $0 \leq i \leq n$, we reach a coloring with six or less colors in $\log^* n$ steps (for all practical purposes this is ≤ 6 or almost constant time).
- Further repetition of the basic coloring will not reduce the number of colors; why? *How to get 3-coloring?*

Make it Work

- Here is the tail end.

  for each m, $3 \leq m \leq 5$, **do**
  for each vertex i such that c(i) = m **pardo**
  Re-color vertex i with the smallest color from {0,1,2} different from its neighbors (obtained from S and P arrays).

- This is done in 3 time units, using O(n) operations in each time unit. Also, this re-coloring is valid since the outer loop is sequential and we never violate rules for coloring.
- Since there are $O(\log^* n)$ steps and each step requires O(n) operations, the algorithm is superfast and *almost* work optimal but not quite so.
- **Exercise:** (a) Write the complete pseudo code of the 3-coloring algorithm formally and write the complete analysis; (b) Consider the ring of 19 nodes, ~~as shown in the class~~; Starting from the beginning, show the color array at each parallel step.

Compute the least significant bit position where two binary numbers differ

- Consider two positive integers x and y – assume $x \geq y$ [if not, interchange them].
- Note that x and y are two positive binary numbers, and we need to find the index j of the rightmost bit in which x and y differ [and we must do it by a constant number of standard operations – the number of operations must be independent of n].
- Set h = x – y and k = h – 1 [So h has a 1 for bit number j, and a 0 for bits of lesser significance; also, h and k agree on the bits of higher significance.]
- Compute $m = h \oplus k$; observe that the integer m is such that it is always equal to 2^j-1; so, compute $j = \lfloor \log_2 m \rfloor$ – assuming your language has that primitive, or in C, int j = ffs (m + 1) [you need to use string.h].
- Example: x = 38 = 100110 and y = 26 = 011010; h = x – y = 12 = 001100, k = h – 1 = 11 = 001011; $m = h \oplus k = 000111 = 7 = 2^j - 1 \Rightarrow j = 2$
- A very interesting and useful resource for C programs is Bit Twiddling Hacks

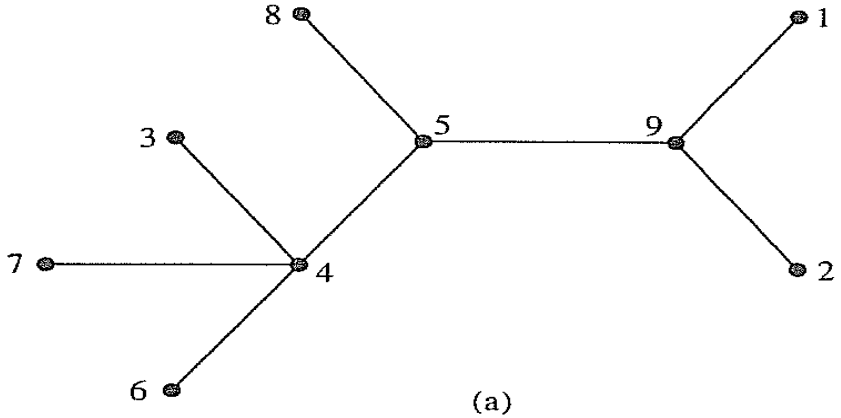
Another Example

x = 58	1	1	0	1	0	0	0	0	
y = 8	0	0	0	0	1	0	0	0	
h = 50	0	0	1	1	0	0	1	0	
k = 49	0	0	1	1	0	0	0	1	
m = 3	0	0	0	0	0	0	1	1	

Tree Computations

Eulerian Tour of a Tree

- Let T = (V, E) be a given tree and let T' = (V, E') be the directed graph obtained from T when each edge (u, v) $\in$E is replaced by two arcs (u, v) and (v, u). Since the in-degree of each vertex of T' is equal to its out-degree, T' is a ***Eulerian graph***; i.e.,, it has a directed circuit that traverses each arc exactly once. It turns out that a Euler circuit of T' can be used for the optimal parallel computation of many functions on T.
- A Euler circuit of T' = (V, E') can be defined by specifying the successor function $\mathcal{S}$, mapping each arc e $\in$E' into the arc $\mathcal{S}$(e) $\in$E' that follows e on the circuit. For each vertex v$\in$V of the tree T = (V, E), we fix an arbitrary fixed ordering on vertices adjacent to v – say, adj(v) = (u_0, u_1, ... ,u_{d-1}), where d is the degree of v. We define the successor of each arc e = (u_i, v) as follows: $\mathcal{S}$((u_i, v)) = (v, $u_{(i+1) \bmod d}$), for $0 \le i \le d-1$, d being the degree of node v [neighbors are numbered 0 through d-1]
- Let us look at an example.



(a)

Given Tree

ordering of nodes

v	$adj(v)$
1	9
2	9
3	4
4	5, 3, 7, 6
5	8, 4, 9
6	4
7	4
8	5
9	5, 2, 1

(b)

Successor function $\mathcal{S}$

Arc	Successor
⟨9, 1⟩	⟨1, 9⟩
⟨9, 2⟩	⟨2, 9⟩
⟨4, 3⟩	⟨3, 4⟩
⟨5, 4⟩	⟨4, 3⟩
⟨3, 4⟩	⟨4, 7⟩
⟨7, 4⟩	⟨4, 6⟩
⟨6, 4⟩	⟨4, 5⟩
⟨8, 5⟩	⟨5, 4⟩
⟨4, 5⟩	⟨5, 9⟩
⟨9, 5⟩	⟨5, 8⟩
⟨4, 6⟩	⟨6, 4⟩
⟨4, 7⟩	⟨7, 4⟩
⟨5, 8⟩	⟨8, 5⟩
⟨5, 9⟩	⟨9, 2⟩
⟨2, 9⟩	⟨9, 1⟩
⟨1, 9⟩	⟨9, 5⟩

(c)

$adj(v) = (u_0, u_1, \ldots, u_{d-1})$,

$\mathcal{S}((u_i, v)) = (v, u_{(i+1) \bmod d})$, for $0 \le i \le d-1$

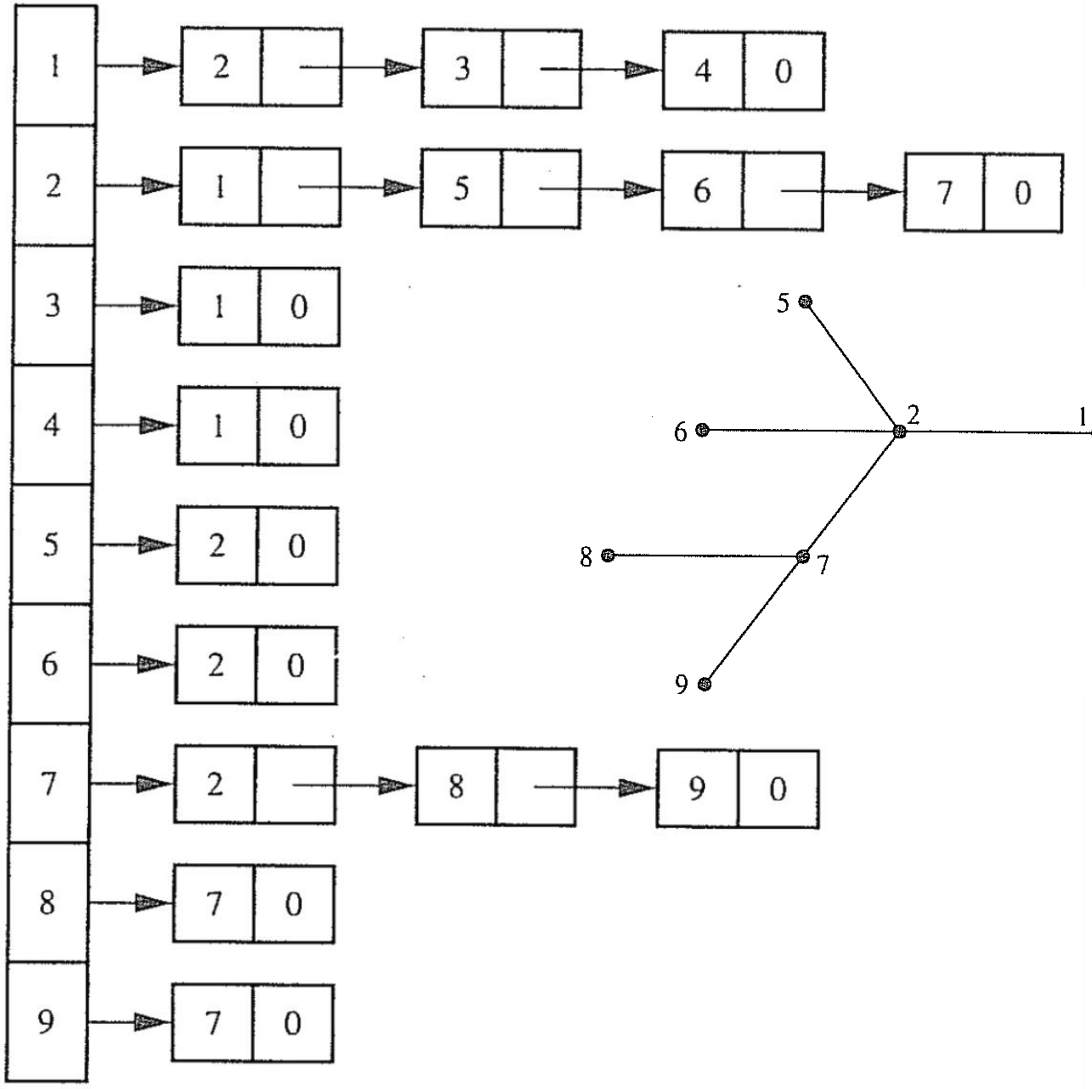
Euler Circuit

⟨9, 1⟩ → ⟨1, 9⟩ → ⟨9, 5⟩ → ⟨5, 8⟩ → ⟨8, 5⟩ → ⟨5, 4⟩ →
⟨4, 3⟩ → ⟨3, 4⟩ → ⟨4, 7⟩ → ⟨7, 4⟩ → ⟨4, 6⟩ → ⟨6, 4⟩ →
⟨4, 5⟩ → ⟨5, 9⟩ → ⟨9, 2⟩ → ⟨2, 9⟩ → ⟨9, 1⟩

Observations

- Given a tree T = (V, E) and an ordering of the set of vertices adjacent to each vertex v ∈ V, the function $\mathcal{S}$ defined previously specifies a Euler circuit in the directed graph T' = (V, E'), where E' is obtained by replacing each e ∈ E by two arcs of opposite directions. T' is called the **Euler Tour** of T. [Note that given the successor function $\mathcal{S}$, each arc *e ∈ E' is assigned a unique* successor and is the successor of a *unique* arc]
- A possible representation of *T* consists of the adjacency lists of the vertices. The adjacency lists L[v]'s for each node v together represent uniquely all arcs in the directed graph T = (V, E'). In addition, the linked list L[v] implies an ordering on the set of vertices adjacent to v, for each v ∈ V.
- See an example.

Example Tree & Adjacency List



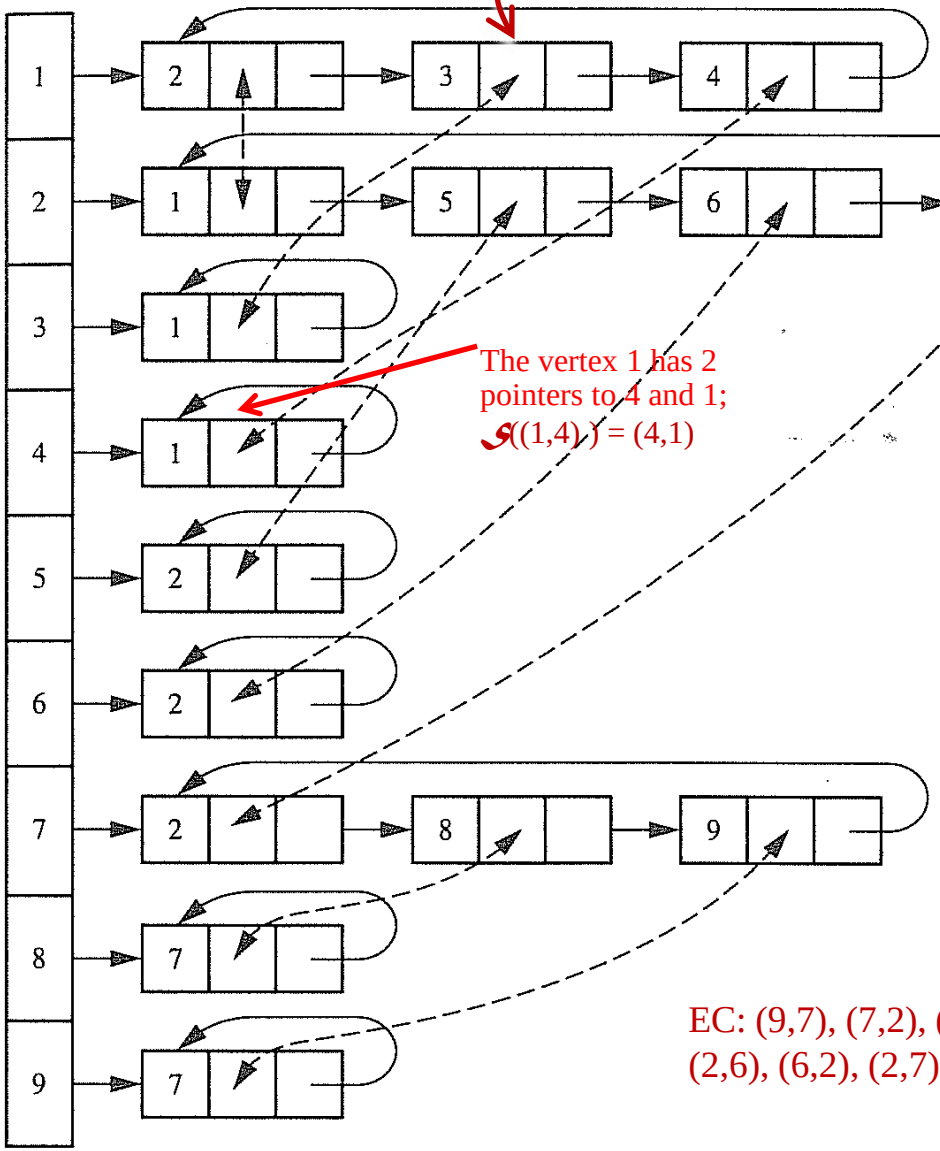
V	Adj(v)
1	2,3,4
2	1,5,6,7
3	1
4	1
5	2
6	2
7	2,8,9
8	7
9	7

Data Structure to store Eulerian Tour

- Remember that $\mathcal{S}((u_i, v)) = (v, u_{(i+1) \bmod d})$, for $0 \leq i \leq d_v - 1$
- To determine the successor function, we need for any list node containing vertex u_i, we need: (1) the arc (u_i, v), and (2) the successor $u_{(i+l) \bmod d}$ of u_i on the list L[v] (which uniquely identifies the arc $(v, u_{(i+l) \bmod d})$
- The successor information is immediately available except when u_i is at the end of the list – make each adjacency list circular.
- To obtain arc (u_i, v), the list node holding vertex u_i in the list L[v] need an additional pointer to the node containing v in $L[u_i]$ (which uniquely represents the arc (u_i, v)).
- It follows that a node on each circular adjacency list consists of a vertex, say u, and two pointers – one is used to deduce the vertex u′ following u on the adjacency list of some vertex v, and hence arc (v, u'); the other pointer is used to obtain the arc (u, v) of T.
- Look at an example.

Circular Adjacency Listing

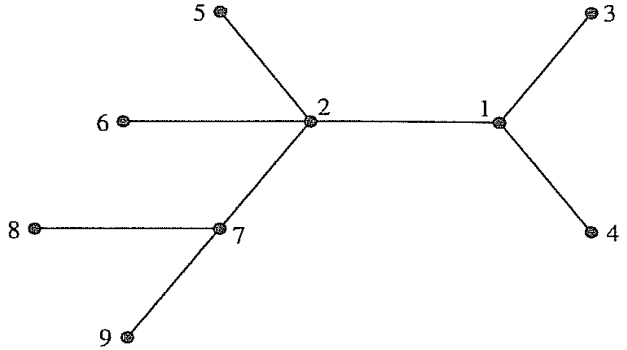
Each node consists of a vertex and 2 pointers



The vertex 1 has 2 pointers to 4 and 1; $\mathcal{S}((1,4)) = (4,1)$

Consider a node in the adjacency listing: it has a vertex 3 and two pointers to vertices 4 and 1 $\Rightarrow$ $\mathcal{S}((3,1)) = (1,4)$ in the Euler circuit.

V	Adj(v)
1	2,3,4
2	1,5,6,7
3	1
4	1
5	2
6	2
7	2,8,9
8	7
9	7



EC: (9,7), (7,2), (2,1), (1,3), (3,1), (1,4), (4,1), (1,2), (2,5), (5,2), (2,6), (6,2), (2,7), (7,8), (8,7), (7,9)

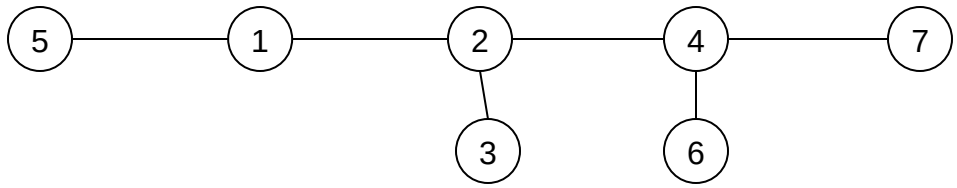
Observations

- Given a tree T = (V, E) defined by the adjacency lists of its vertices with 2 additional pointers, we can construct an Euler circuit in T' in O(1) time using O(n) operations, where |V| = n.
- How? Consider an arbitrary node of the adjacency lists. This node holds a vertex, say u, and a pointer to its successor holding, say vertex *u'*, *and* another pointer to a node holding a vertex *v*. *Clearly, (u, v) is an arc such that* $\mathcal{S}((u, v)) = (v, u')$. *For each node, this operation can be carried out in O(1)* time, since the pointers in the node holding vertex *u uniquely identify arcs (u, v) and (v, u'). Therefore, the Euler circuit can be generated in O(1) time*, using a linear number of operations for the given input representation.
- Note that there are 2n directed edges in T', hence 2n nodes in the adjacency list, hence we need 2n processors – no concurrent memory access – EREW PRAM model.
- We never needed the tree to be rooted.
- So, we assume, whenever Euler tour is needed, the tree is represented by the set of circular adjacency lists with the additional pointers.

Rooting a Tree

- Rooting a given tree at an arbitrary node , say r, means that we compute a predecessor function p(v), $v \neq r$, when the given tree T is rooted at node r [p(r) = r].
- We compute Euler tour of T as before (using the successor function). Let the adjacency list of node r is $L[r] = (u_0, u_1, ..., u_{d-1})$. Break the tour by setting $\mathcal{S}(u_{d-1}, r) = 0 \Rightarrow$ we now have a directed Euler path EP, in T' that begins at r, visits each arc exactly once, and ends at r. Observe:
  - EP is an ordered set of edges, which is nothing but traversing the tree in depth first order starting from the node r.
  - The directed edge (p(u), u) always appear on the list EP before the arc (u, p(u)) for all nodes u when the tree is rooted at r.
  - There are 2n edges in the EP where n is the number of nodes in the tree

Example & Algorithm



Root the tree at node 2. Compute EP by setting S((4,2)) = ∅

arcs	(2,1)	(1,5)	(5,1)	(1,2)	(2,3)	(3,2)	(2,4)	(4,6)	(6,4)	(4,7)	(7,4)	(4,2)
I. W.	1	1	1	1	1	1	1	1	1	1	1	1
PS	1	2	3	4	5	6	7	8	9	10	11	12

I.W. denotes the initial weight attached to each edge; PS is the prefix sum on the initial weights!

PS (1,2) > PS (2,1) $\Rightarrow$ 2 = p(1); PS (1,5) < PS(5,1) $\Rightarrow$ 1 = p(5); PS (2,3) < PS (3,2) $\Rightarrow$ 2 = p(3);
PS (2,4) < PS(4,2) $\Rightarrow$ 2 = p(4); PS(4,6) < PS (6,4) $\Rightarrow$ 4 = p(6); PS (4,7) <PS (7,4) $\Rightarrow$ 4 = p(7);

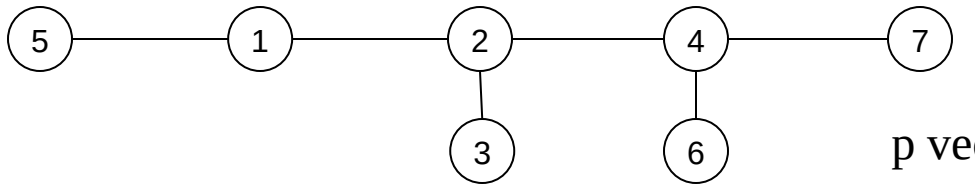
Algorithm Outline:

1. Identify last vertex u in L[r], set $\mathcal{S}$(u,r) = 0.
2. Assign a weight of 1 to each arc; do a parallel prefix sum on the list of arcs.
3. for each arc (x, y): PS (x, y) < PS (y,x) $\Rightarrow$ x = p(y) [Assign a PE to each arc]

Claim: Rooting of a tree can be done in O(log n) time using O(n) operations on EREW PRAM.

Post Order Numbering

- Post order Numbering of a rooted tree is numbering the vertices as they appear in a post order traversal of the tree (left → right, root); this is also known as post order ranking. Note:
  - The left to right ordering of the children of a root is the one implied by the Euler tour EP of the tree.
  - EP visits each node v several times – first time via the edge (p(v),v) and the last time via the edge (v, p(v)), after visiting all the descendents of node v.



The tree is rooted at node 2.

p vector = [2 2 2 2 1 4 4]

arcs	(2,1)	(1,5)	(5,1)	(1,2)	(2,3)	(3,2)	(2,4)	(4,6)	(6,4)	(4,7)	(7,4)	(4,2)
I. W.	0	0	1	1	0	1	0	0	1	0	1	1
PS	0	0	1	2	2	3	3	3	4	4	5	6

Initial weights: for each v ≠ r, w(v, p(v)) = 1 and w (p(v), v) = 0;
Post Order #: for each v ≠ r, post (v) = PS (v, p(v)) and post (r) =n;

In this example: post [ ] = (2, 7, 3, 6, 1, 4, 5); e.g., post (7) = PS (7, p(7)) = PS ((7,4)) = 5

Note: Parallel run time is O(log n) using O(n) operations in EREW model.

Other Optimal Computations in a Tree in O(log n) time on EREW PRAM

- Vertex **Level**: Compute the level(v) of each vertex v in a rooted tree T, which is the distance (number of edges) between v and the root r. We again use the Euler path of T rooted at r. We assign w(p(v), v)) = +1, and w(v,p(v))) = -1, and perform parallel prefix on the list defining the Euler path. Then, we set level(v) to be equal to the prefix sum of (p(v), v).
- **Size** of a node (# of nodes in the subtree rooted at this node): Let size(v) be the number of vertices in the subtree rooted at v. Consider the Euler path EP defined by the successor function s. We apply the parallel prefix algorithm to the list EP using the weight w(p(v), v) = 0 and w(v, p(v)) = 1, for each vertex $v \neq r$. Then, size(v) is equal to the difference between the prefix sums of (v,p(v) and (p(v), v), since all the arcs appearing between (p(v), v) and (v,p(v)) on the Euler path EP belong to the subtree rooted at v ($v \neq r$).
- Pre-order Numbering:
- Symmetric Order numbering [binary tree]:

Expression Evaluation (Tree Contraction)

Expression Evaluation

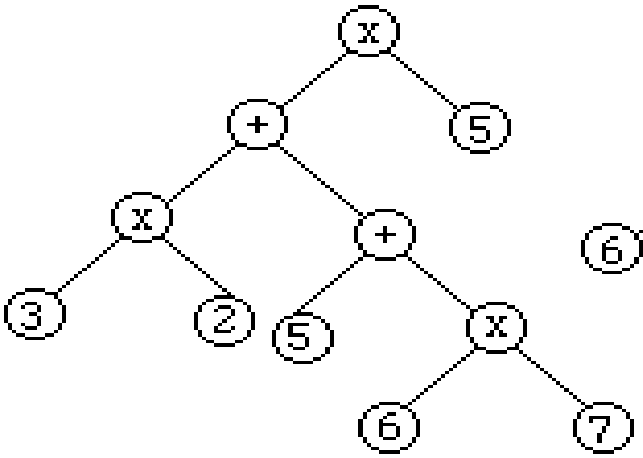
To add the numbers 2, 1, 3, 2, 1, 3, 2, 1 we can write a sequential program

x1 = 2; x2 = x1+1; x3 = x2+3; x4 = x3+2;
x5 = x4+1; x6 = x5+3; x7 = x6+2; x8 = x7+1

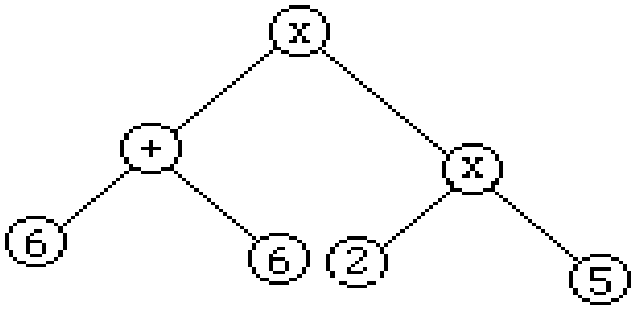
Compilers often produce code that consists of a tree representation. In this tree, called an expression tree, leaves represent numbers or variables, and internal nodes represent operations. Expression trees are, by definition, full binary trees – each internal node has exactly two children.

Expression Tree Evaluation: Given an expression tree, compute in parallel the value it represents.

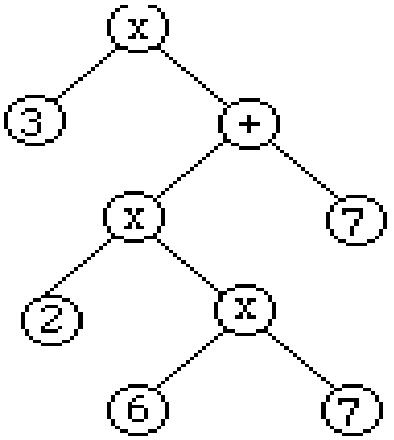
Examples



(3*2+(5+6*7))*5,
= 265



((6+6)*(2*5)),
= 120



(3*((2*(6*7))+7)
= 273

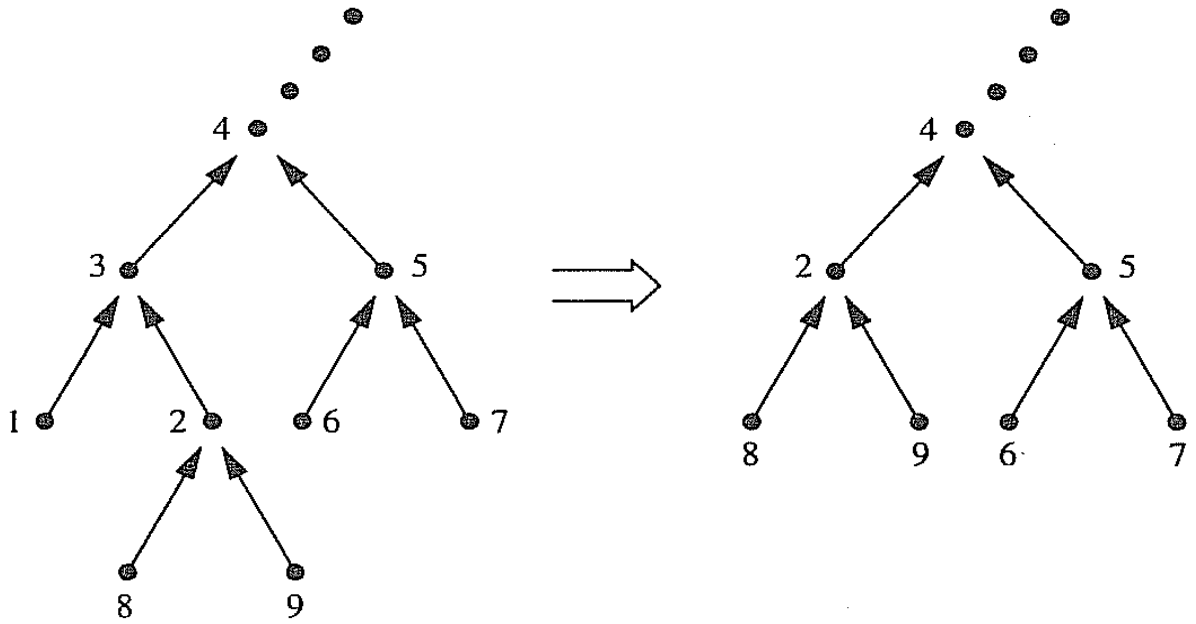
Note:

- One could think that parent nodes with children two leaves can start processing independently becoming leaves themselves, until the whole tree has been contracted to a single node. If the expression tree is *balanced*, as the middle one above, this approach is good: it takes O(height) = O(log n). But if the tree is not balanced, like the right one, then it may take O(n) in the worst case, which is expensive.
- We may be interested to know the expressions computed at the subtrees rooted at every internal node. Giving all the sub expressions of the left tree in postorder, we have: 3, 2, 6, 5, 6, 7, 42, 47, 53, 5, 265.

Tree Contraction

Given a leaf node u such that p(u) ≠ r, the rake operation applied at u consists of removing u and p(u) (parent of u) from T and connecting the sibling of u, denoted by sib(u), to p(p(u)).

How to use Rake to Contract the Tree



- We show what happens when the rake operation is applied to node 1of the tree of the left-hand side. This operation results in removing nodes 1 and 3, and in making node 4 the parent of node 2, as shown in the right-hand tree.
- Our algorithm for tree contraction uses the rake operation as the primitive operation to reduce a given input binary tree into a three-node tree consisting of the root and two leaves. The main technical difficulty lies in avoiding the concurrent rake of two leaves whose parents are adjacent. We can avoid that difficulty by carefully applying the rake operation to nonconsecutive leaves as they appear from left to right.

Tree Contraction Algorithm

- Input: (1) A rooted binary tree T such that each vertex has exactly two children, and (2) for each vertex u different from the root, the parent p(u) and the sibling sib(u).
- Output: T is contracted to a three-node binary tree.

begin

S1: Label the leaves consecutively in order from left to right, excluding the leftmost and the rightmost leaves, and store the labeled leaves in an array A of size n.

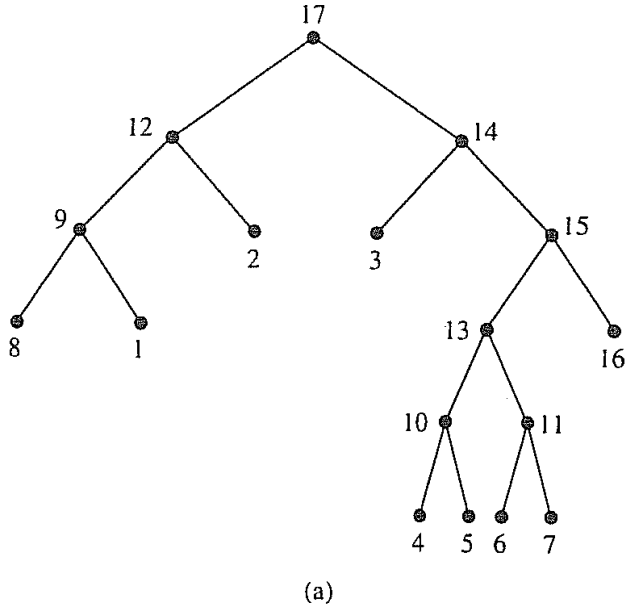
S2: **for** $\lceil \log (n + 1) \rceil$ iterations **do**

1. Apply the rake operation concurrently to all the elements of A_{odd} that are left children.
2. Apply the rake operation concurrently to the rest of the elements in A_{odd}.
3. Set $A := A_{even}$.

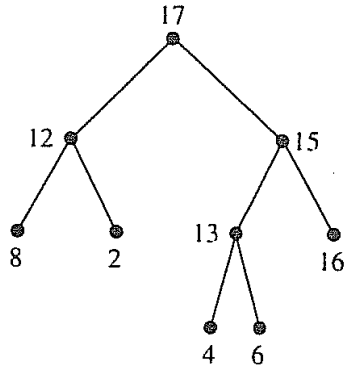
end

Note: For a given array A, A_{odd} is the subarray of A consisting of the odd indexed elements of A (e.g., a_1, a_3, a_5, ...). We define the subarray A_{even} similarly. Once A and its length are given, A_{odd} and A_{even} and their lengths can be determined in O(1) time, using a linear number of operations.

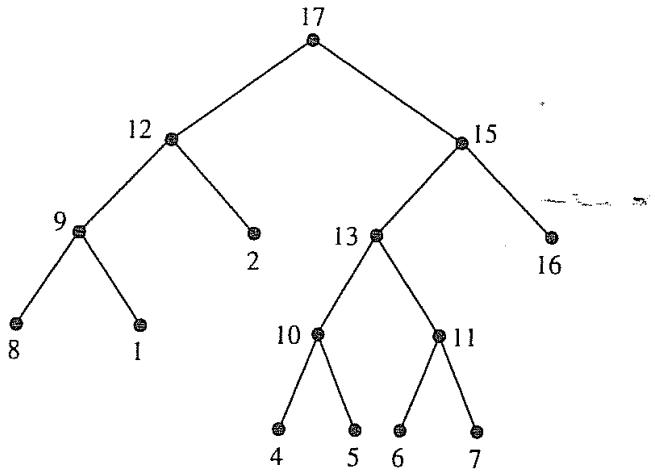
An Example



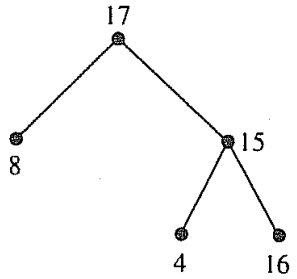
(a)



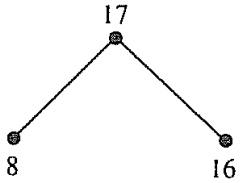
(c)



(b)



(d)



(e)

- Here, A = (1, 2, 3,4, 5, 6, 7), A_{odd} = (1, 3, 5, 7), and A_{even} = (2, 4, 6).
- During the first iteration of the for loop, the rake operation is applied to only vertex 3; we get the tree in (b).
- Step 2.2 of the first iteration results in applying rake to vertices 1, 5, and 7, giving us the tree shown in (c).
- At the end of the first iteration, we have A = (2, 4, 6).
- The second iteration of the loop rakes leaves 2 and 6 (step 2.2), resulting in the tree (d).
- Finally, the third iteration shrinks the tree to the three-node tree (e).

Analysis of the Algorithm

The tree-contraction algorithm correctly contracts the input binary tree into a three-node binary tree. This algorithm can be implemented on the EREW PRAM in O(log n) time, using O(n) operations, where n is the number of vertices in the input tree.

- Note that, whenever the rake operation is applied concurrently to several leaves, the parents of any two such leaves are not adjacent, and hence rake is applied correctly.
- At the end of each iteration of the main loop, #leaves decreases from m to $\lfloor m/2 \rfloor$, where m is the number of leaves at the beginning of the iteration. Hence, after $\lceil \log n \rceil$ iterations, all leaves disappear except the leftmost and rightmost leaves.
- Step 1 is implemented by using the Euler-tour technique; leaves appear from left to right on the Euler path of T.
- The rake operations are applied in parallel. Hence, steps 2.1, 2.2 and 2.3 take O(1) time.
- The number of operations required by each iteration is O(|A|), where |A| is the current size of the array A. Since $|A_{even}| \leq |A|$, the total number of operations is $O(\Sigma(n/2^i)) = O(n)$. Thus, the overall time is O(log n), and the total number of operations is O(n).

Note: we do not really need to introduce the subarrays A_{odd} and A_{even} in since the leaves can be stored consecutively in an array, and at each iteration we know the exact positions of the leaves to be removed.

Lowest Common Ancestors

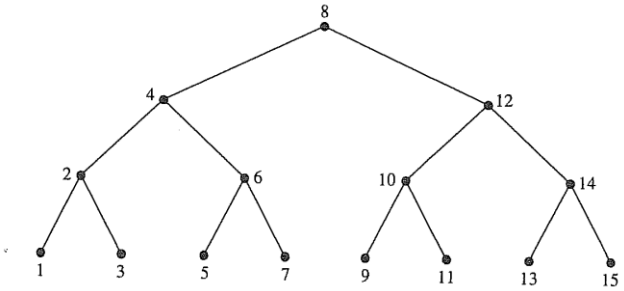
- The lowest common ancestor of two vertices u and v of a rooted tree is a node w that is an ancestor to both u and v and is farthest from the root. For a rooted tree T = (V, E), the problem of finding the lowest common ancestor of an arbitrary pair of vertices u and v, denoted by lca(u, v), arises in many situations.
- This is a very common type of query in many applications that need be answered for many pairs dynamically during executions of applications.
- We need to pre-process the tree in such a way that the LCA query can be answered in constant or O(1) time.
- There are many ways to solve this problem – we will use an EP technique.

Two Special Cases

- When the tree T is a simple path, computing the distance of each vertex from the root allows us to answer any LCA(u, v) query in constant time by comparing the distances of u and v from the root.
- When the tree is a complete binary tree, then compute the in-order number of each node and number the nodes accordingly; each vertex number is a $\lceil \log_2 n \rceil = m$ bit binary number. LCA(u,v), where $u = u_1\ u_2\ ..\ u_m$ and $v = v_1\ v_2\ ..\ v_m$ and where i is the minimum integer such that $j < i \Rightarrow u_j = v_j$ and $u_i \neq v_i$, can be computed as $LCA(u,v) = u_1\ u_2\ ..\ u_{i-1}$,100 ..0

Consider nodes 9 = $(1001)_2$ and 13 = $(1101)_2$;

LCA (9, 13) = $(1100)_2$ = 12

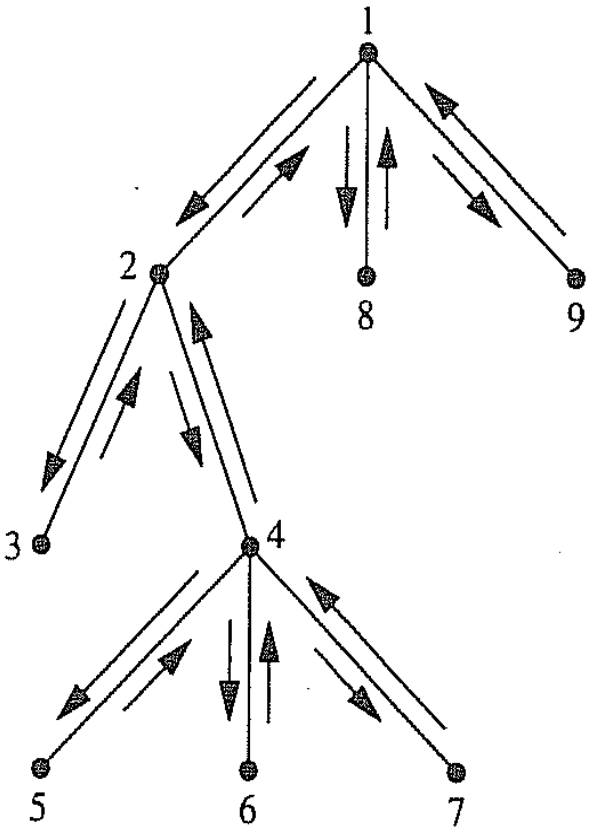


- One strategy to solve the general LCA problem is to mapping the given tree into a logarithmic depth complete binary tree.
- Another approach is to reduce the problem to a range minima problem – we need to look at certain properties of trees.

Reduction Process

Given a tree, consider the EP, the Euler Path Array A as the corresponding ordered vertices with the root inserted at the beginning and each ordered pair (u,v) replaced by v. From array A, compute array B by taking the level of each element of A; note that |A| = 2n – 1 if T has n nodes.

- Consider a natural left to right ordering of the vertices for the EP.
- EP = (1,2)(2,3)(3,2)(2,4)(4,5)(5,4)(4,6) .... (9,1)
- We need 2 more arrays L & R: L(v) = index of the leftmost appearance of v in A and R(v) = index of rightmost appearance of v in A.
- How to get L and R? Consider $v \neq r$; for array A, $A_i = v$ is the leftmost appearance of v iff $B_{i-1} = B_i - 1$; Also, $A_i = v$ is the rightmost appearance of v in array A iff $B_{i+1} = B_i - 1$. Note: Both L and R can be computed in parallel in O(1) time using O(n) work.



v	1	2	3	4	5	6	7	8	9
level	0	1	2	2	3	3	3	1	1
L	1	2	3	5	6	8	10	14	16
R	17	12	3	9	6	8	10	14	16

$A = (1, 2, 3, 2, 4, 5, 4, 6, 4, 7, 4, 2, 1, 8, 1, 9, 1)$

$B = (0, 1, 2, 1, 2, 3, 2, 3, 2, 3, 2, 1, 0, 1, 0, 1, 0)$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

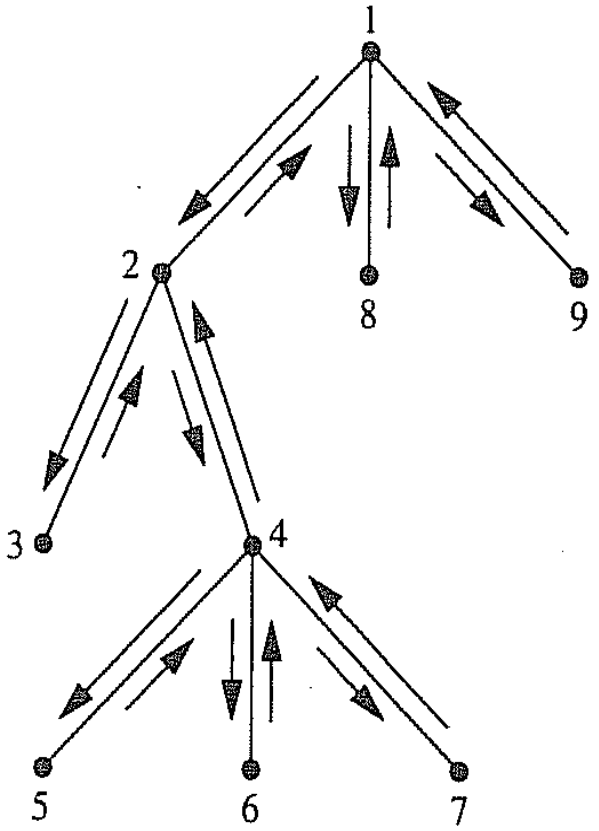
Observations

- Given a rooted tree T = (V, E), let A, level(v), L(v) and R(v), for v $\in$V, be as defined previously. Let u and v be two arbitrary distinct vertices of T. Then, the following statements hold:
  1. u is an ancestor of v iff L(u) < L(v) < R(u).
  2. u and v are not related; that is, u is not a descendant of v and v is not a descendant of u, iff either R(u) < L(v) or R(v) < L(u).
  3. If R(u) < L(v), then LCA(u, v) is the vertex with the minimum level over the interval [R(u), L(v)].
- Proof:
  1. If u is an ancestor of v, the EP [depth-first traversal of T from r], visits u before v $\Rightarrow$ L(u) < L(v) and the subtree rooted at v is completely visited before the last visit of v, i.e., L(v) < R(u). Conversely, if u is not an ancestor of v, then L(u) < L(v) means that subtree rooted at u is completed before the first visit to node v, i.e., R(u) < L(v) [contradiction]
  2. R(u) < L(v) $\Rightarrow$ u is not an ancestor of v and R(v) < L(u) $\Rightarrow$ v is not an ancestor of u.
  3. We can easily check that if R(u) < L(v), then all vertices whose levels appear in the closed interval [R(u), L(v)] are either vertices appearing on the path between u and v or their descendants $\Rightarrow$ vertex with minimum level must be the LCA(u, v).

Thus, computing LCA(u, v) for two arbitrary vertices in a tree T is as follows: We need an efficient algorithm to preprocess the array B = level(A) so that the following range-minimum query can be processed in O(1) sequential time. Suppose B is an array of size $n=2^k$; for two arbitrary given indices, i, j, $1 \le i < j \le n$, we need to determine the minimum of the sub-array $[b_i b_{i+1} \ldots b_j]$ in O(1) or constant time. Note that elements of B are nonnegative, not necessarily distinct.

Example

v	1	2	3	4	5	6	7	8	9
level	0	1	2	2	3	3	3	1	1
L	1	2	3	5	6	8	10	14	16
R	17	12	3	9	6	8	10	14	16



Example 1: We need LCA (3,7). L(3) = 3, R(3) = 3, L(7) = 10, R(7) = 10; R(3) < L(7) $\Rightarrow$ we need the minimum of the $[b_3 \ldots b_{10}]$ = [2 1 2 3 2 3 2 3] which is 1 corresponding to the vertex 2 [found from array A]

Example 2: We need LCA (6, 8). L(6) = 8, R(6) = 8, L(8) = 14, R(8) = 14; R(6) < L(8) $\Rightarrow$ we need the range minimum $[b_8 .. b_{14}]$ = [3 2 3 2 1 0 1] which is 1 corresponding to the vertex 1 [found from array A]

$A = (1, 2, 3, 2, 4, 5, 4, 6, 4, 7, 4, 2, 1, 8, 1, 9, 1)$

$B = (0, 1, 2, 1, 2, 3, 2, 3, 2, 3, 2, 1, 0, 1, 0, 1, 0)$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Range Minima

Range Minima

- The problem is to preprocess an array B of size $n=2^k$ such that for two arbitrary given indices, i, j, $1 \leq i < j \leq n$, we need to determine the minimum of the sub-array $[b_i b_{i+1} \ldots b_j]$ in O(1) or constant time.
- A natural way to proceed is to build a complete binary tree on the elements of B – the internal nodes will contain some info about the leaf nodes rooted at this internal node – such that each internal node v of T holds some information about the array determined by the leaves in the sub tree rooted at v.
- Given two indices i and j, we can determine the LCA v of the leaves of T holding b_i and b_j in O(1) sequential time, since T is a complete binary tree*. If all leaves of the sub tree rooted at v correspond exactly to the sub array $\{b_i, b_{i+1}, \ldots, b_j\}$, then it is sufficient to store the minimum element in the node v. But that's not true in general; why?

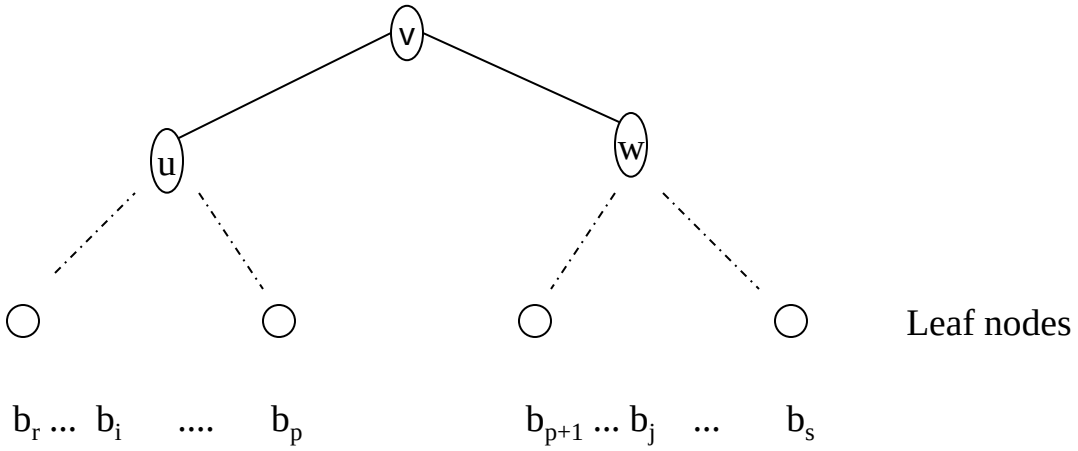
*[Remember the earlier special case – Consider in order numbering of the nodes; if there are n nodes in the tree, each node number can be encoded by $\lceil \log_2 n \rceil$ bits; let $u = u_1 u_2 \ldots u_{\lceil \log n \rceil}$ and $v = v_1 v_2 \ldots v_{\lceil \log n \rceil}$; let i be the minimum integer such that $j < i \Rightarrow u_j = v_j$ and $u_i \neq v_i$; then, LCA (u, v) = $u_1 u_2 \ldots u_{i-1} 100 \ldots 0$].

Approach

- In general, the set of leaves that are rooted at node v is of the form $\{b_r \ldots b_i \ldots b_j \ldots b_s\}$, $r \leq i \leq j \leq s$. Say, *v* has a left child u and a right child w. Thus, the sub-arrays of leaves associated with nodes v, u, w, are:

$$B_v = \{b_r \ldots b_i \ldots b_j \ldots b_s\},$$

$$B_u = \{b_r \ldots b_i \ldots b_p\},\ B_w = \{b_{p+1} \ldots b_j \ldots b_s\} \text{ for some } p,\ i \leq p < j$$



Approach (contd.)

- The min$\{b_i\, b_{i+1} \ldots b_j\}$, needed at node *v* is actually the minimum of two elements: (1) minimum of the suffix $\{b_i \ldots b_p\}$ of B_u and (2) minimum of the prefix $\{b_{p+1} \ldots b_j\}$ of B_w. Thus we need to store at each internal node *v* the suffix minima and prefix minima of sub array associated with the node *v*.
- The prefix minima of B are the elements of the array $\{c_1 \ldots c_n\}$ such that $c_i = \min\{b_1 \ldots b_i\}$ for $1 \leq i \leq n$. Similarly, the suffix minima of the array $\{d_1 \ldots d_n\}$ such that $d_i = \min\{b_{n-i+1} \ldots b_n\}$ for $1 \leq i \leq n$. Both these arrays (prefix and suffix minima) can be computed in *O(n)* operations by using the prefix sum algorithm.
- Given an array B of size $n = 2^h$, we need a complete binary tree with auxiliary variables P(h,j) and S(h,j), $0 \leq h \leq log_2 n$, $1 \leq j \leq n/2^h$ such that P*(h,j)* and S*(h,j)* represent respectively the prefix and suffix minima of the sub array defined by the leaves of the sub-tree rooted at *(h,j)*.

PRAM Code

S1: **for** $1 \leq j \leq n$ **pardo**

P(0,j) = B(j); S(0,j) = B(j)

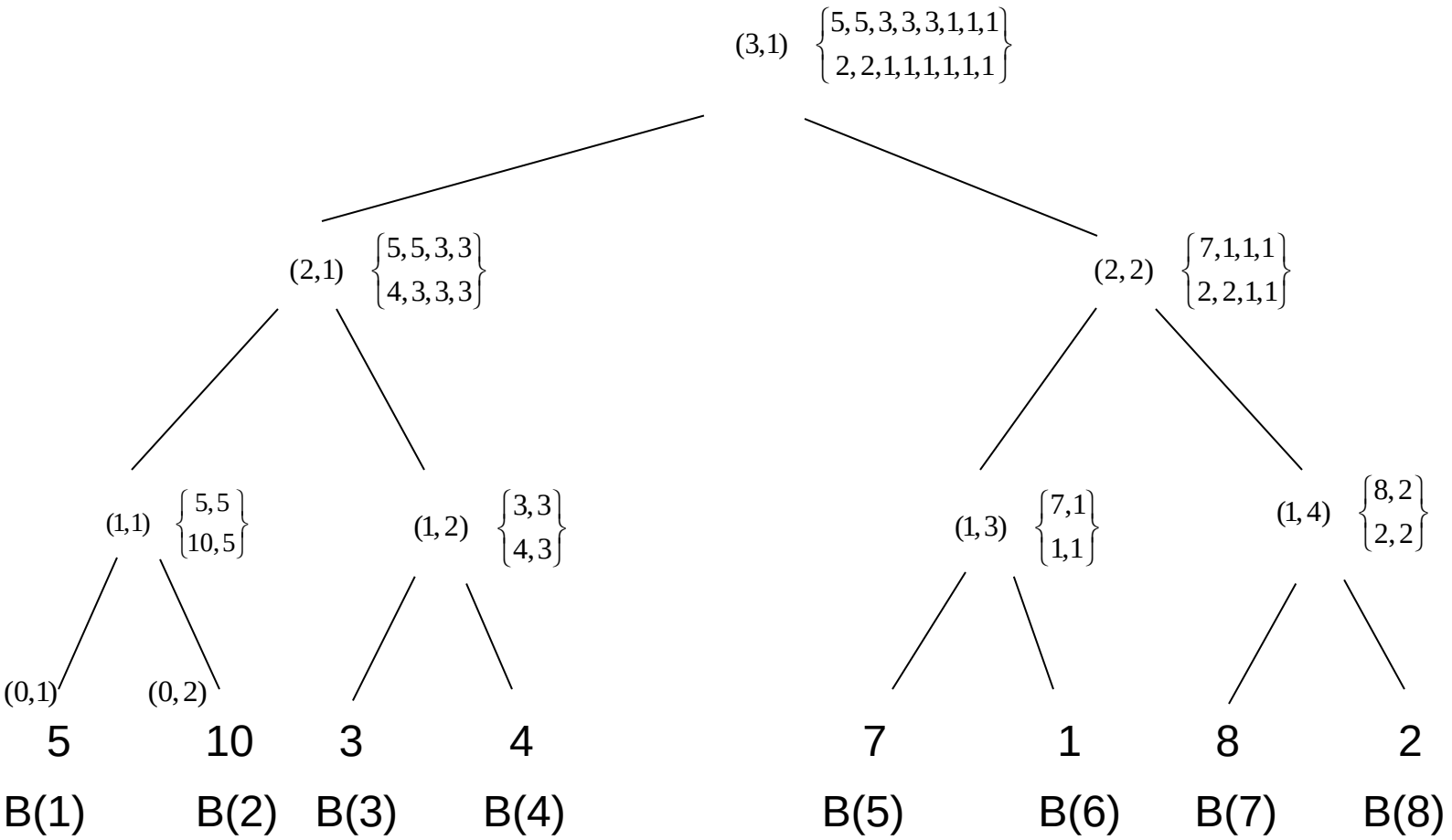
S2: **for** $h = 1$ to $log_2 n$ **do**

for $j = 1$ **to** $n/2^h$ **pardo**

merge P*(h-1, 2j-1)* **&** P(*h-1, 2j*) **in** P(*h,j*);

merge S(*h-1, 2j-1*) & S*(h-1, 2j)* **in** S(*h,j*)

Example



P(3,1) is computed by copying P(2,1) into the 1st half of P(3,1) and copying P(2,2) into the 2nd half; then each element α of the 2nd half of P(3,1) is replaced by the minimum of α and the last element of P(2,1), i.e., min{α, 3) in this case. The other P and S arrays are generated in the same manner.

How to use it?

- We need to answer range minimum query in the interval [*i*,*j*] i.e., we need to report the minimum of the interval $\{b_i, b_{i+1}, \ldots, b_j\}$. Let *v* = LCA(*i*,*j*); node v can be computed in O(1) time and let *u* and *w* be the left and right children of *v*; then, the desired result is the minimum of two elements: the suffix minimum corresponding to *i* in the S array of *u* and the prefix minimum corresponding to *j* in P array of *w*. [How do we compute v in O(1) time with the given data structures? Need to write the code.]
- Say i = 2 and j = 5; note, B(2) = 10 and B(5) = 7. The LCA(2,5) is the root node (3,1) whose children are nodes (2,1) and (2,2). Note S(2,1) = (4,3,3,3) and P(2,2) = (7,1,1,1). Thus answer to the query is the minimum of the second element of S(2,1) and the first element of P(2,2) – that is, minimum of 3 & 7 which is 3.

Time for Preprocessing

- Parallel time needed to preprocess the array B is $\mathcal{O}(log_2 n)$; the algorithm uses $\mathcal{O}(nlog_2 n)$ operations; each query can be answered in $\mathcal{O}(1)$ sequential time.
- **Merge sub arrays at each level of the tree**: Size of each array P(h,j) is 2^h; Merging P(h-1,2j-1) and P(h-1,2j) into P(h,j) consists of copying P(h-1,2j-1) into the 1st half of P(h,j) and replacing each element α of P(h-1,2j) with the minimum of α and last element of P(h-1,2j-1); this can be done in parallel in $\mathcal{O}(1)$ time using $\mathcal{O}(|P(h,j)| = 2^h)$ operations. Merging of S arrays are similar; thus, each level of the tree needs $\mathcal{O}(2^h \times n/2^h) = \mathcal{O}(n)$ operations [At each level h of the tree, there are $n/2^h$ internal nodes] . Thus, total run time is $\mathcal{O}(log_2 n)$ doing total $\mathcal{O}(nlog_2 n)$ operations.
- **Given two indices *i* and *j*, compute LCA (*i,j*)** = v, say. Node v in the complete binary tree is to be located by two integers: level and serial number at that level. Note that in-order number of node B(i) is $2i-1 = p$, and that of B(j) is $2j-1 = q$ say. Compute LCA (p,q) in constant time and the both sides of the separating position index will provide the two integers needed to locate the LCA. [The number of zeros on the rhs of the separating line is the level and the binary value of the lhs + 1 is the serial number].

Test

- Write Something; Pradip Srimani MWrite Something; Pradip Srimani Monai, Rono