EC504 ALGORITHMS AND DATA STRUCTURES
SPRING 2021 MONDAY & WEDNESDAY
2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

Simple Data Structures: Abstract Data Type (ADT)

- In Object Oriented Programming data and the operations that manipulate that data are grouped together in classes
- Abstract Data Types (ADTs) or data structures or collections store data and allow various operations on the data to access and change it
- High level languages like C++, Java, Python often provide implementations of ADTs

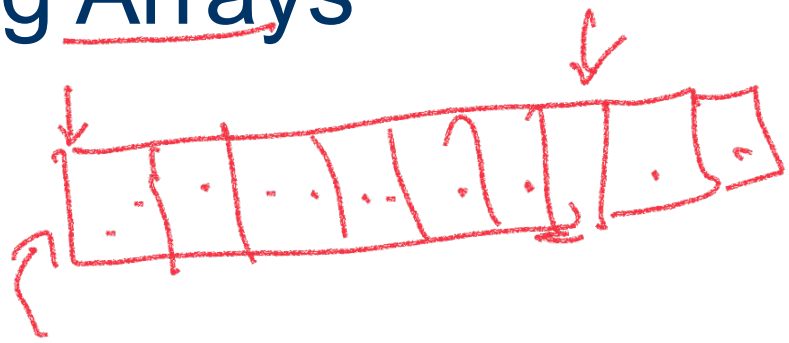
The List ADT

- List of size N: $A_0, A_1, \ldots, A_{N-1}$
- Each element A_k has a unique position in the list
- Elements can be arbitrarily complex
- Operations
  - Depend on what the list is for
- Implementation: either as arrays or as linked lists

Sample List (C++) ADT

```
class List {
  public:
  List();                          // constructor
  List(const List& list);          // copy constructor
  ~List();                         // destructor
  List& operator=(const List& list);  // assignment operator

  bool empty() const;              // boolean function
  void addHead(double x);          // add to the head
  double deleteHead();             // delete, get head

  void addEnd(double x);           // add to the end
  double deleteEnd();              // delete, get end

  bool findNode(double x);      // search for a given x
  void deleteNode(double x);       // delete node with value x

  int length() const;              // count the number of elements

};
```

Lists Using Arrays



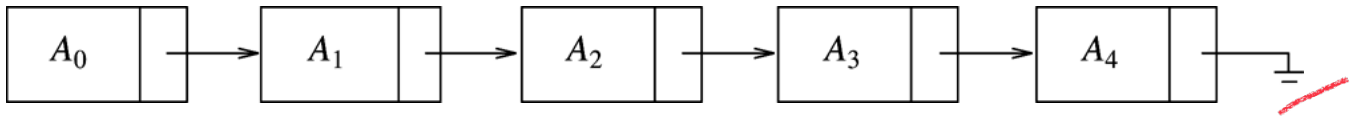
- Operations
  - addHead(X) – O(N)
  - deleteHead() – O(N)
  - addEnd(X) – O(1)
  - deleteEnd() – O(1)
  - findNode(X) – O(N)

slow

O(1) if Circular.

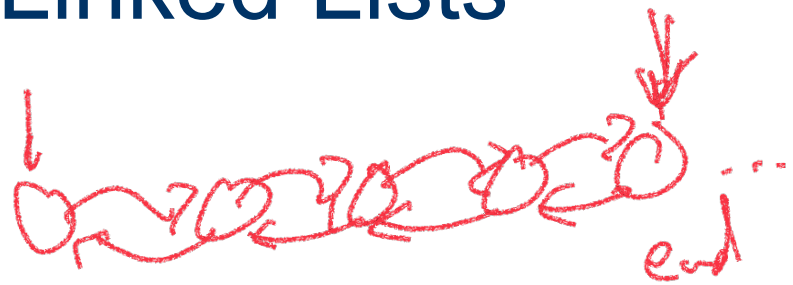
Linked Lists

- Elements not stored in contiguous memory
- Nodes in list consist of data element and next pointer



struct node
* node, data;

Lists Using Linked Lists

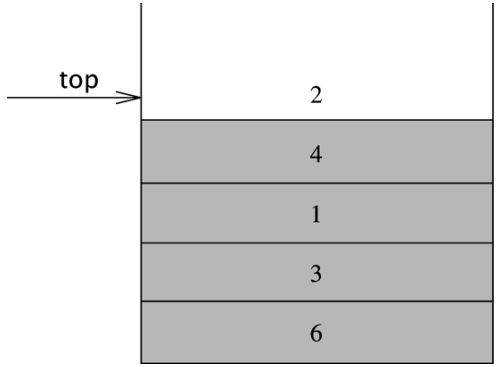


- Operations
  - addHead(X) – O(1)
  - deleteHead() – O(N)
  - addEnd(X) – O(N)
  - deleteEnd() – O(N)
  - findNode(X) – O(N)

- Can modify using doubly-linked lists, pointers to end
  - Make addEnd(X), deleteEnd O(1).

List Applications: Stacks

- Stack is a list where insert and remove take place only at the “top”
- Operations
  - Push (insert) element on top of stack
  - Pop (remove) element from top of stack
  - Top: return element at top of stack



- LIFO (Last In First Out) queue
- Stack is a list, any list implementation will do, but will have different advantages

Stack ADT

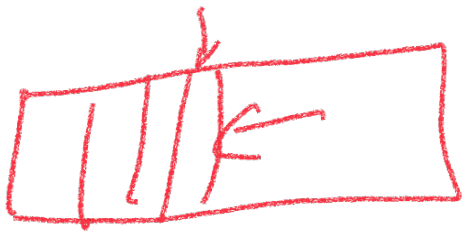
```
class Stack {
  public:
      Stack();                        // constructor
      Stack(const Stack& stack); // copy constructor
      ~Stack();                       // destructor

      bool empty() const;
      void push(const double x);
      double pop();                   // change the stack

      double top() const;  // keep the stack unchanged

  private:
};
```

Stack using linked lists

```
struct Node{
   public:
         double data;
         Node* next;
};

class Stack {
   public:
         Stack();                    // constructor
         Stack(const Stack& stack); // copy constructor
         ~Stack();                   // destructor

         bool empty() const;
         void push(const double x);
         double pop();               // change the stack

         bool full(); // unnecessary for linked lists
         double top() const;         // keep the stack unchanged
         void print() const;

   private:
         Node* top;
};
```

Stack using arrays


```
class Stack {
public:
      Stack(int size = 10);                  //
constructor
      ~Stack() { delete [] values; }         // destructor

      bool empty() { return top == -1; }
      void push(const double x);
      double pop();

      bool full() { return top == maxTop; }
      double top();
      void print();
private:
      int maxTop;          // max stack size = size - 1
      int top;             // current top of stack
      double* values;      // element array
```



Array versus Linked List Implementation

- push, pop, top are all O(1) constant-time operations in both array and linked list implementation
  - For array implementation, the operations are performed in very fast constant time
- Linked List structures are larger - need pointers
- Array allocate space contiguously - useful for cache, etc.
- Arrays may have to be resized (lots of data movement…) but ***amortized analysis*** says worst case is still O(1)!

O(n) once every n.

Stack Application: Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart (e.g. MATLAB, code syntax checkers, …)
  - e.g. [( )] is legal, but [( ]) is illegal
- Algorithm

  (1) Make an empty stack.

  (2) Read characters until end of file

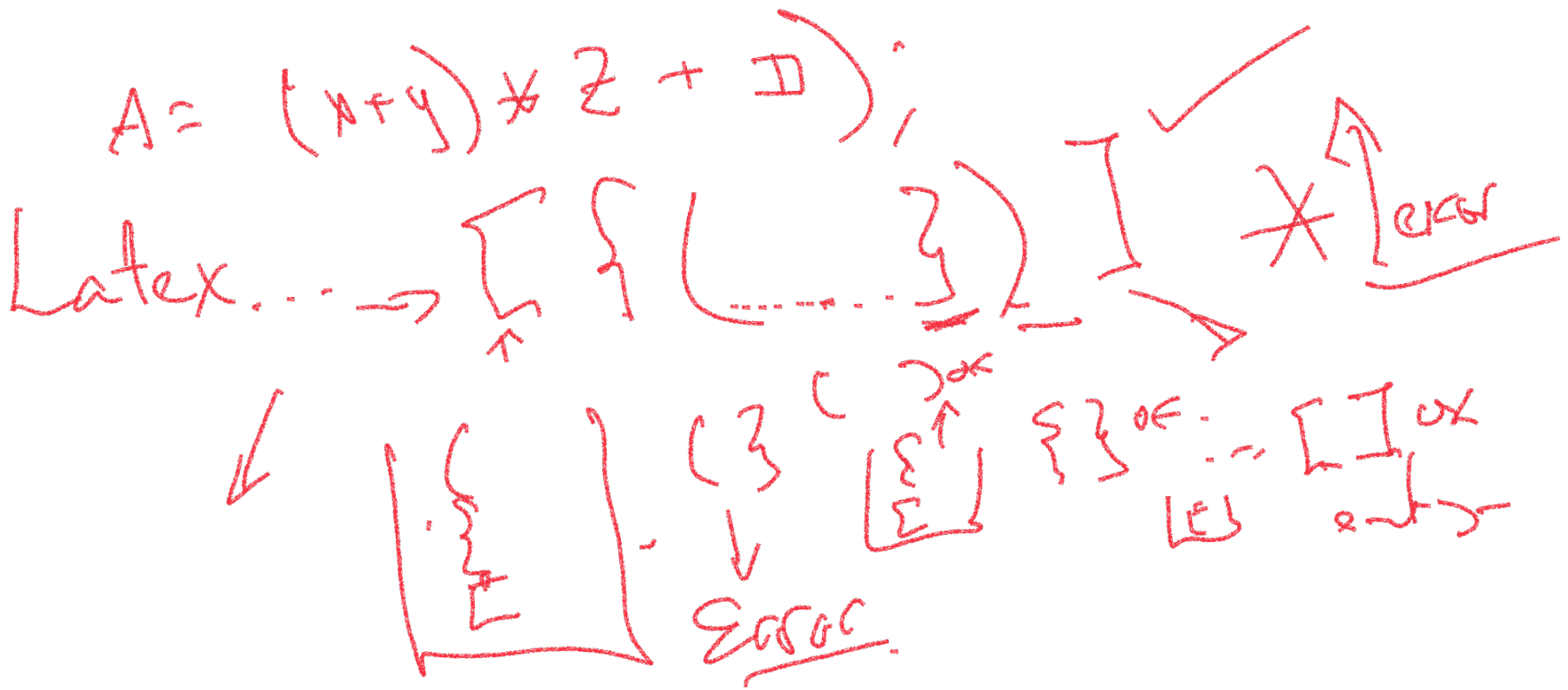
  i. If the character is opening symbol, push onto the stack

  ii. If closing symbol and stack is empty, error

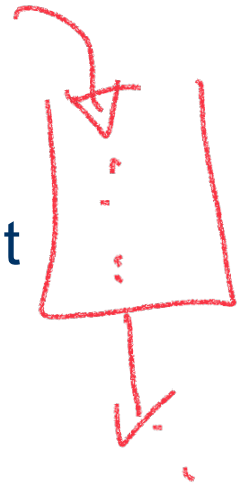
  iii. Otherwise, pop stack. If symbol popped is not corresponding opening symbol, error

  (3) At end of file, if stack is not empty, error

- Other applications
  - Function calls, recursion, …
  - Back/Forward stacks on browsers
  - Undo/Redo stacks in Excel or Word



Queue



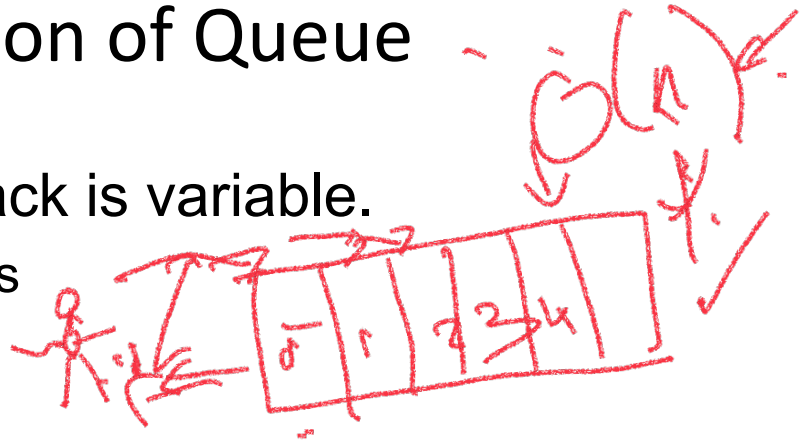
- Queue is a list where insert takes place at back, but remove takes place at the front
- Operations
  - Enqueue (insert) element at the back of the queue
  - Dequeue (remove and return) element from the front of the queue
- FIFO (First In First Out)
- Applications: simple job scheduling, graph traversals, priority queues, routing, printer queues, I/O streams

Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Need to have fast access to beginning and end of queues
  - Dequeue at front
  - Enqueue at back
- Similar advantages for arrays vs linked lists
  - But there are different ways to do it with arrays which can result in much data movement, O(n) computation!

Array implementation of Queue

- Naive: front is always position 0, back is variable.
  - Dequeue is O(n)! Must move positions



- Better way: Use circular array
  - When an element moves past the end of a circular array, it wraps around to the beginning, e.g. size 9, top 5, bottom 8
  - Enqueue(4); $000007963 \rightarrow 400007963$ (top 5, bottom 0)
  - Detect an empty or full queue: Use a counter of the number of elements in the queue.
  - O(1)
  - Only issue is fixed size… reallocate if full. Amortized cost O(1).

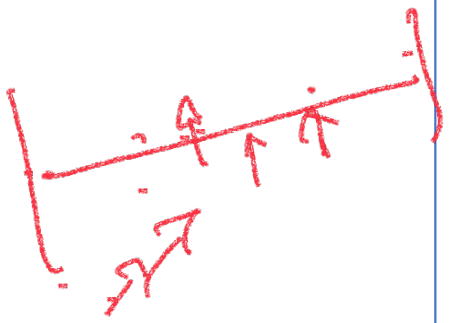
Problem of Interest: Search

- Problem: finding an element among a group of elements
  - Data structure issue: what is a good way of storing that data so search is fast?

- Issues:
  - Will the data change? How often? How?
  - Do we search for arbitrary elements by value? By relative value (e.g. largest, smallest, 5th largest?) By order of insertion (e.g. last in, first out (LIFO) or first in, first out (FIFO)?

Sorted Lists

- If list changes rarely, best approach may be to store sorted list
  - Best implementation: arrays
  - With array implementation, many steps are fast
    - Find k-th largest: O(1)
    - Find x: Binary search (bisection) O(log(n))
    - Find median: O(1)
  - But rare operations such as insert, delete, or change_value are O(n)
    - Insertion sort
  - We need a search algorithm and a sorting algorithm
    - Focus of modern search algorithms



Binary Search of Sorted List

- int a[0], a[1],a[2],a[3],.... a[m],.... a[N-2],a[N-1]


```
i= 0; j= N-1; m = N/2
 while(b!=a[m] && i!=j ){
            if(b>a[m])    i = m+1;
           if(b<a[m])    j = m-1;
            m =   (j-i)/2 + i;  }
if(b==a[m])) “found it” else “not found”
```

Choose mid point

$$T(N) = T(N/2) + c_0 \Longrightarrow T(N) \in O(\log(N))$$

Sorting Algorithms

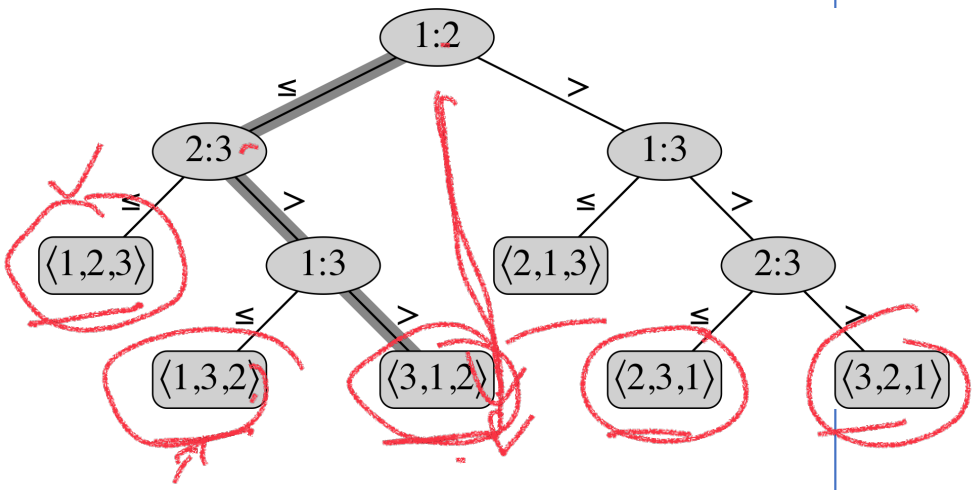
- There are lots of them…
- Most are useless
- Two classes: Comparison-Based vs non-comparison based
  - non-comparison: bucket sort, radix sort, … Work when number range is discrete and known (e.g. 16-bit integers)
    - Can be O(n), but constants depend on value range
    - If values are distinct, constant is O(log(n))!
  - Mergesort, Insertion Sort are comparison-based: Steps compare values.

Sorting Algorithms

- Theorem: All comparison-based algorithms are $\Omega(n \log(n))$
- Proof outline: Comparison-based algorithms are similar to binary decision trees
  - Each comparison has binary branch
  - Tree ends in leaves
  - #leaves = #permutations (n!)
  - Height of binary tree must be at least $log_2(n!) \in \Omega(n \log(n))$
  - More on binary trees later

$n! \approx O(n^n)$

e.g. Sort [9,1,2]



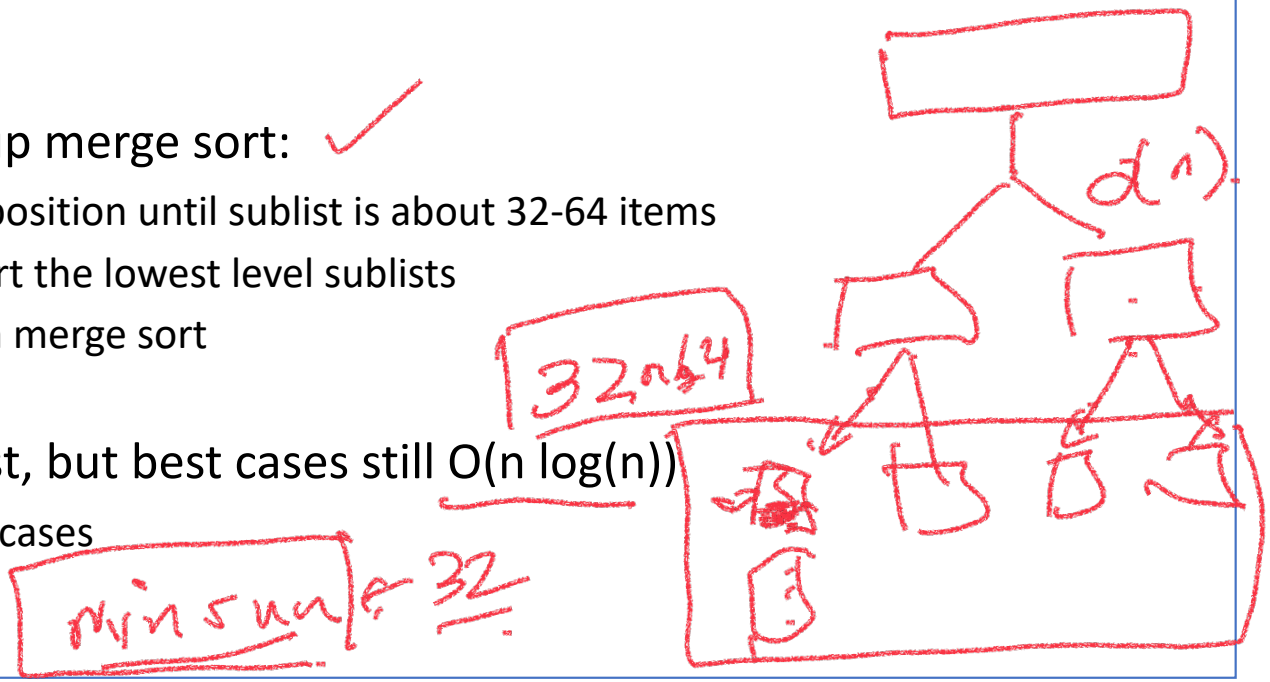
Comparison-based Sorting Algorithms

Name	Best	Average	Worst	Stable	Extra Memory	Complexity
Quicksort	$n \log(n)$	$n \log(n)$	n^2	Depends	$\log(n)$	Small
Merge Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$	Yes	n	Small
Heapsort	$n \log(n)$	$n \log(n)$	$n \log(n)$	No	1	Small
Insertion Sort	n	n^2	n^2	Yes	1	Small
Shell Sort	n	$n(\log(n))^2$	$n(\log(n))^2$	No	1	Small
Bubble Sort	n	n^2	n^2	Yes	1	Small
Selection Sort	n^2	n^2	n^2	Depends	1	Small
Timsort	n	$n \log(n)$	$n \log(n)$	Yes	n	Medium

Would like optimal best case and optimal worst case! Fast & robust

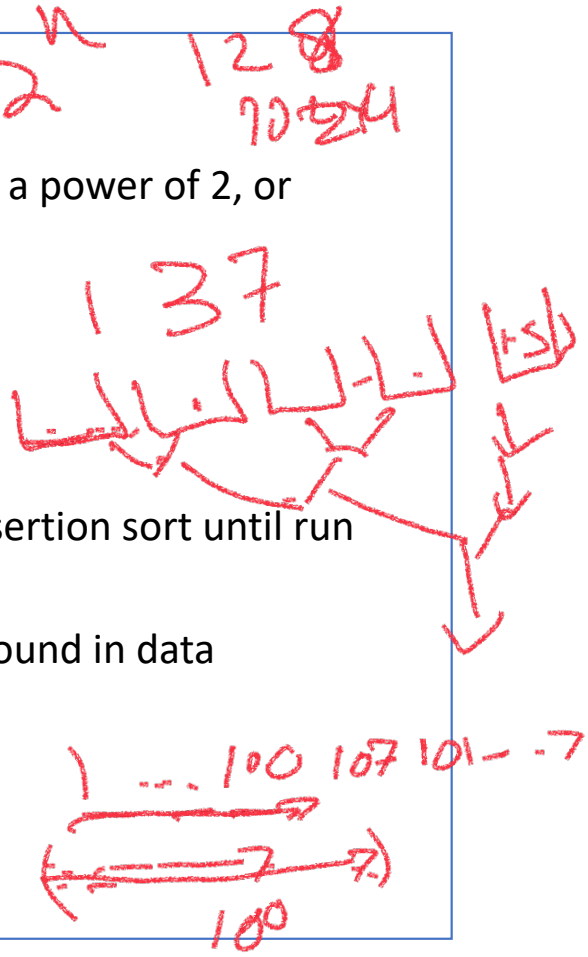
Timsort (Tim Peters, 2002)

- Currently used in Python, Java, other places
  - Combination of Insertion Sort and Merge Sort plus clever recognition of good cases
  - Example of what makes good algorithms: robust in worst-case, but able to take advantage of good structure
- One way to do speed up merge sort:
  - Do merge sort decomposition until sublist is about 32-64 items
  - Do insertion sort to sort the lowest level sublists
  - Merge the results as in merge sort
- Result O(n log(n)) worst, but best cases still O(n log(n))
  - Does not exploit good cases



Key ideas in Timsort

- Run: a sorted sublist of elements.
  - A run is the longest non-decreasing or decreasing sequence.
- minrun: minimum size for run, typical 32 to 65, selected so N/minrun is a power of 2, or perhaps a little smaller
- Give a list of numbers of length N:
  - Identify minrun based on N
  - Starting at the beginning of a list, look for a run
    - If run length is less than minrun, add next elements using insertion sort until run length is minrun
    - Run length can be much greater than minrun if structure is found in data
  - Merge consecutive runs using a merge stack
    - Avoid regression, merge with only one temporary array
    - Advanced idea: galloping during merging (not essential…)



Timsort Illustrated

Sort 1 2 3 4 11 7 5 6 8 9 10

https://www.chrislaux.com/timsort.html

Determine minrun in 2, 3, 4: N = 11, so minrun = 3.

Runs: [1 2 3 4 11] [7 5] —> Now run less than minsize and is decreasing —> reverse run andinsert next element 6 —> [5 6 7].

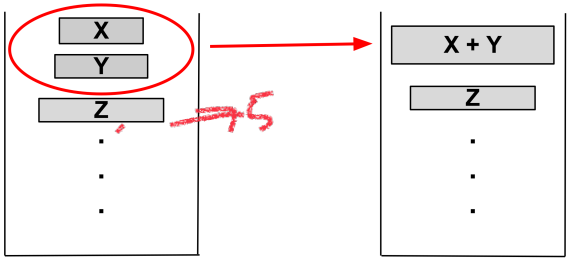
Runs: [1 2 3 4 11] [5 6 7][8 9 10]

Merge stack: Key properties:

a. $\#Z > \#X + \#Y$

b. $\#Y > \#X$

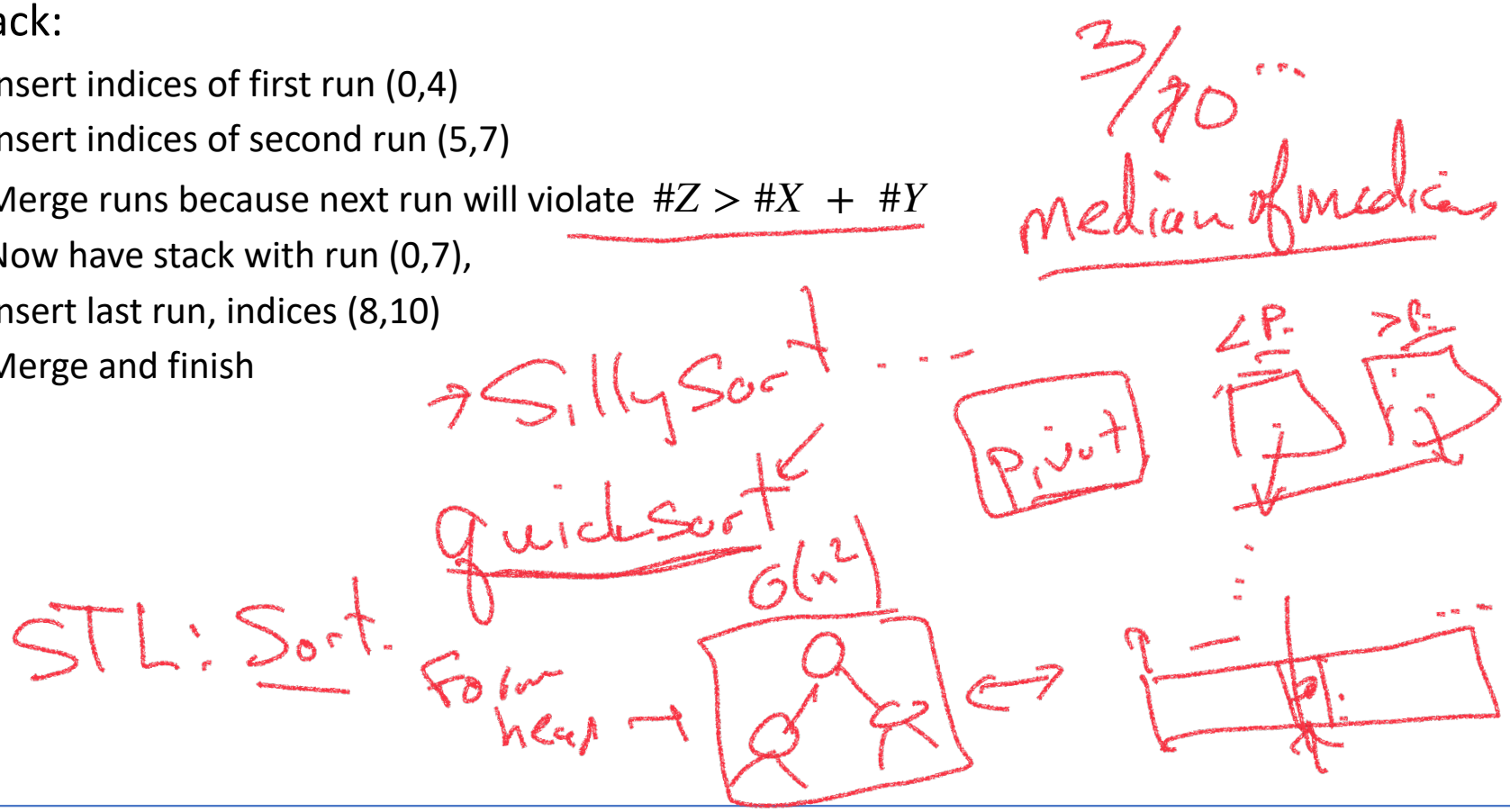
Merge if these are violated.



Timsort Illustrated (2)

Stack:

- Insert indices of first run (0,4)
- Insert indices of second run (5,7)
- Merge runs because next run will violate $\#Z > \#X + \#Y$
- Now have stack with run (0,7),
- Insert last run, indices (8,10)
- Merge and finish



Search with Dynamic Data

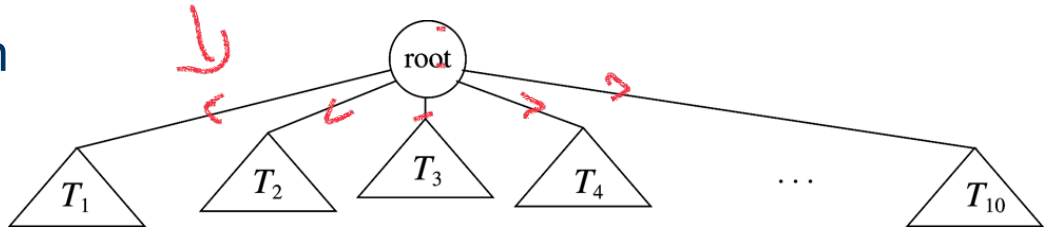
- Problem: finding an element among a group of elements
  - Frequent insertions, frequent deletions, modifications of data
- Operations of interest:
  - Find, insert, delete, find minimum, find maximum

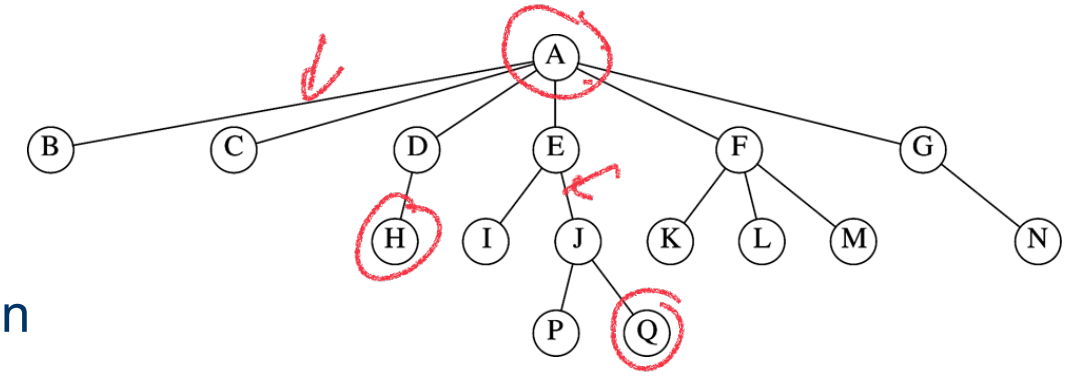
- If we use a sorted list:
  - Takes O(n log(n)) to form sorted list initially from n elements
  - Takes O(n) to insert: Either because of time to find or data movements
  - Takes O(n) to delete: Either because of time to find or data movements

- Alternative: Search trees

Background on Trees

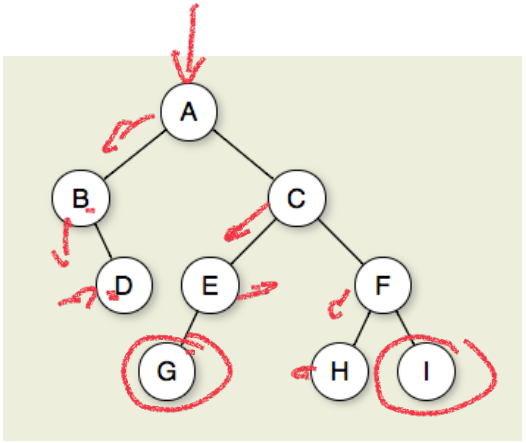
- Many definition of trees
  - Root node with trees as children
  - Connected, undirected graph with no cycles
  - …
- Important concepts
  - Node: a structure with data and pointers to children
  - Edge: a link from a node to a node that is one of its children

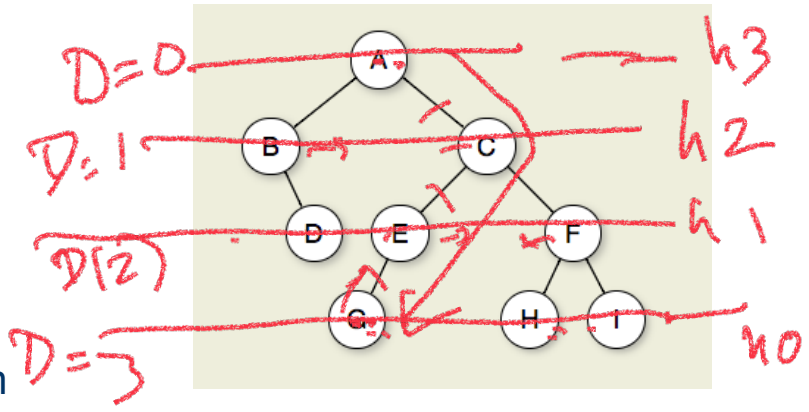




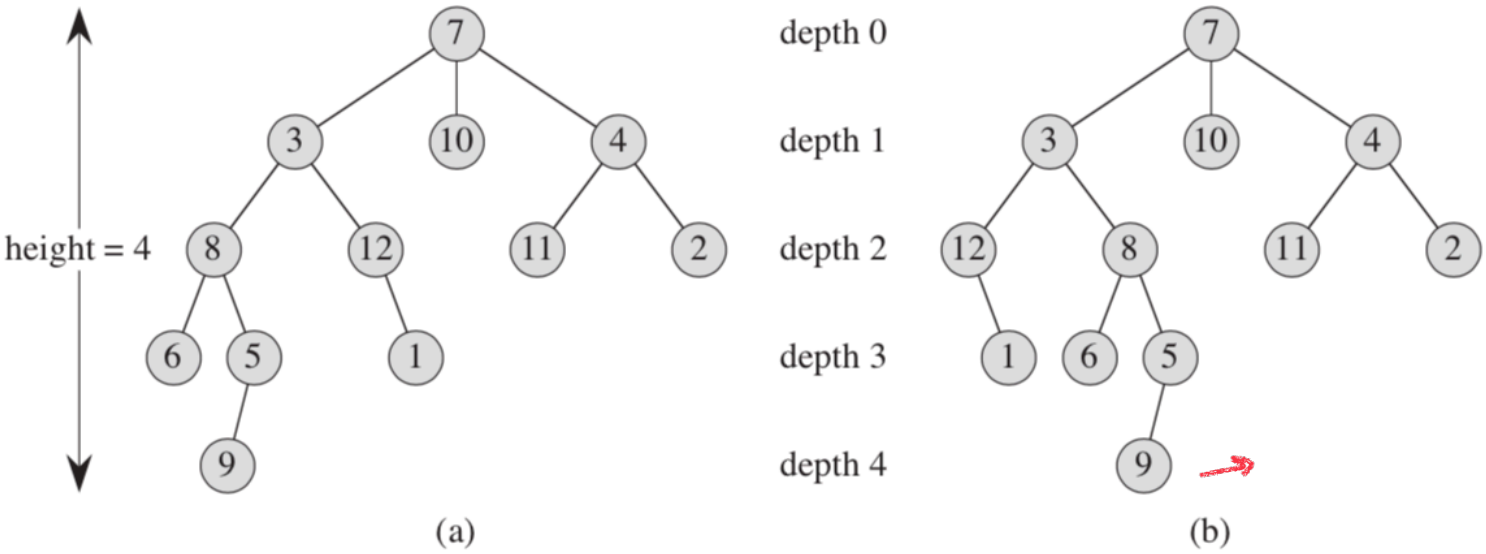
Trees

- **Node**: an element of the tree. Contains data, pointers to other nodes
- **Root**: The entry node of the tree. Has no parent
- **Edge**: the link from a node to a child node
- **Child(X)**: A node that is linked from node X
- **Parent(X)**: The node in the tree that has X as its child
- **Sibling(X)**: a node that has the same parent as X
- **Leaf**: A node with no children nodes
- **Internal node**: has at least one child
- **Path(n_1, n_k)**: a sequence $n_1, n_2, \ldots, n_k$ so n_{i+1} is child of n_i
- **Depth(n)**: number of edges on path from root to n.
- **Height**: maximum depth of any leaf
- **Ancestor**: n ancestor of m if there is path from n to m
- **Descendant**: m is a descendant of n if there is path from n to m



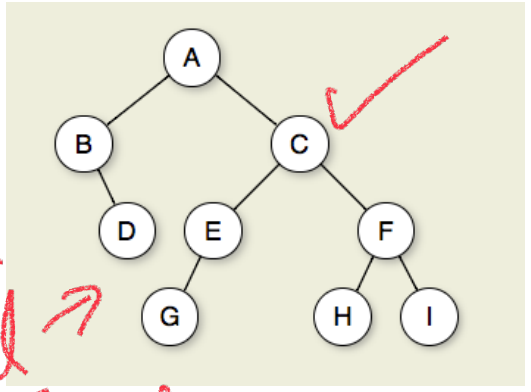


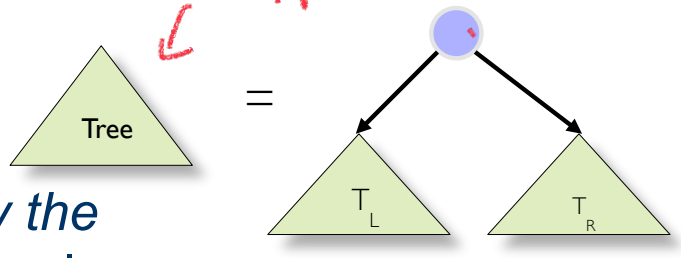
Height vs Depth of “nodes”

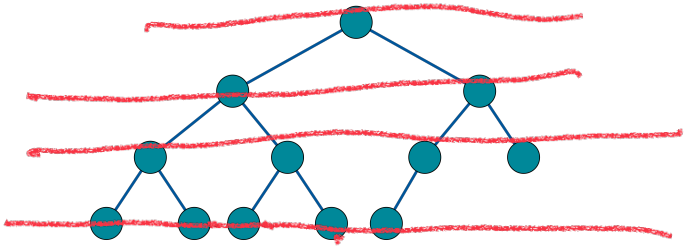


Special Case: Binary Trees

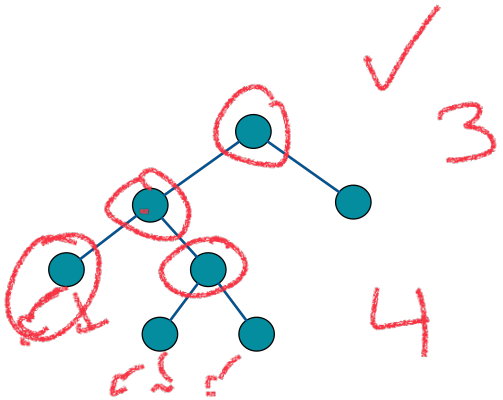
- Binary Tree: Every node has at most 2 children (0, 1 or 2)
- Recursive definition: A binary trees is null, or a singe node with a
  Right and Left Child that is a binary tree!
- Special cases:
  - Full Binary Tree: Every node has 0 or 2 children
  - Perfect Binary Tree: all interior nodes have two children and all leaves have the same depth
  - Complete Binary Tree: every level, *except possibly the last*, is completely filled, and all nodes in the last level are as far left
    as possible



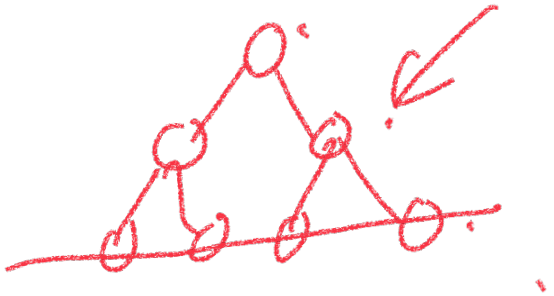




Complete binary tree



Full binary tree

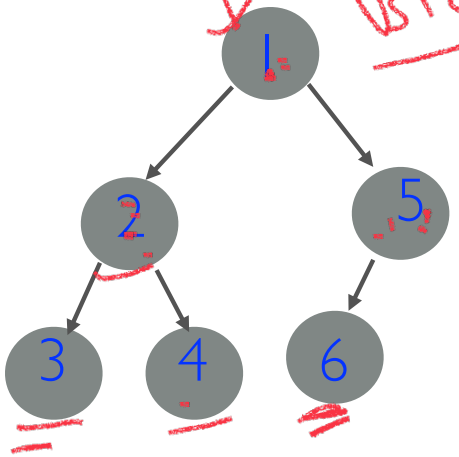


Perfect binary tree

- Simple results:
  - Full binary tree: number of leaves = number of internal nodes + 1
  - In non-empty binary tree, number of null children = number number of nodes in trees
  - In perfect binary tree, number of nodes is $2^{height+1} - 1$
    - For n nodes, height = $log_2(n+1) - 1$

Binary tree traversals

```
■Preorder:    Print [Tree]{
                    Print root;
                    Print Tree[LeftTree];
                    Print Tree:[RightTree];
               }
■Inorder:     Print [Tree]{
                    Print  Tree[LeftTree];
                     Print root;
                    Print Tree:[RightTree]
               }
■Postorder:   Print [Tree]{
                  Print  Tree[LeftTree];
                 Print Tree:[RightTree]
                  Print root;
                 }
```



Pre: 1→2→3→4→5→6

In: 3→2→4→1→6→5

Post: 3→4→2→6→5→1