Monday, July 20

Second exam

Strategy to delete last node

```
declare 2 node references: current, previous;
  /* then handle special cases first */
just return (i.e., do nothing) if list is empty;
set first to null and return if just one node;
  /* otherwise traverse list to find second-to-last node */
point previous at first node;
point current at previous.next;
while (current.next does not refer to null)
    advance both previous and current references;
  /* finally, set link of second-to-last */
set previous.next = null; // old last node is garbage collected
/* Done. */
```

Efficiency of list functions

- If singly-linked list:
  - Insert/delete first – O(1)
  - Insert/delete last/middle – O(n)
  - Find value – O(n)
  - Retrieve/set i^{th} item – O(n)
- Compare to array:
  - Insert/delete first/middle, and find value – O(n)
  - Insert/delete last – O(1) – unless resize, then O(n)
  - Retrieve/set i^{th} item – O(1) – the array's strong point

Improved lists

- Some improvements can increase usefulness
  - e.g., circular list to solve Josephus problem
  - e.g., generalized lists – actually are lists of lists
- Some improvements aim to speed up operations
  - e.g., maintain a separate reference to last item
    - Now O(1) complexity to access last
    - Still O(n) to delete last – Why?
  - Double-linked list is even better (next slide)
- Trade-offs – the usual: speed ↔ space ↔ effort

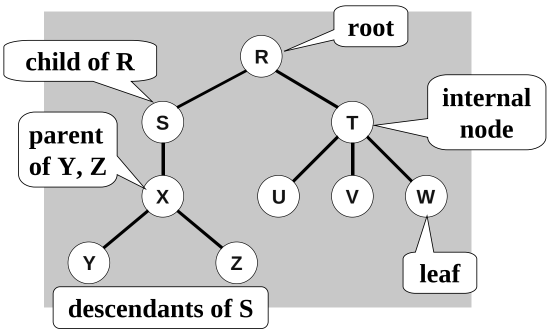
Double-linked lists

- A Node – has links to next *and* previous nodes
- A List – has references to both first and last nodes
- More work to implement most operations though
  - Twice as many links to worry about for all cases
  - More special cases to consider – 2^{nd} and penultimate
- But easy to traverse backwards
  - Also O(1) to delete last, easy to insert before a node, …
- Sentinel nodes – a trick to eliminate special cases
  - First and last nodes hidden from client – never empty!
  - See java.util.LinkedList.java

Implementing priority queues

- Way 1 – *unsorted* array (or ArrayList / Vector)
  - insert – easy: add item as last array element
  - remove – harder: search for highest priority item, and move last element to emptied slot
  - Insert is O(1), remove is O(n)
- Way 2 – *sorted* list
  - insert – some work: search for right position – O(n)
  - remove – easy: remove the first item – O(1)
- Way 3 – a type of tree called a heap – later

Trees – some terminology



More tree terms and concepts

- Every tree has exactly one root
  - Root is `null` for an empty tree
  - But each node really is the root of its own subtree
- Starting from the root, there is exactly one path to each node (would be a graph if could be more than one path)
- The depth of a node is the length of the path from the root to this node (a.k.a., "level")
  - Depth of the root is 0
  - Path length is the number of edges between two nodes
- The height of a tree equals the greatest node depth in the tree (the height of an empty tree is -1)

Basic tree operations

- Some operations are common to all trees
  - Height of tree, count items, clear items, isEmpty
  - Insert item, find item, delete item, depth of item
  - Also ways to visit items (traverse) in various orders
- Rules for some operations vary by tree type
  - Some trees have ordering principles
  - Some trees have structure principles
  - Some trees cannot store duplicates
    - Such a tree qualifies as a Set

Tree ADTs vary widely

- Behaviors depend on the type of tree
- Efficiency of operations also varies
  - Depends on rules, and often on tree structure
- Structures vary too
  - Shape may be fixed, or allowed to vary only slightly
  - Or shape can change dramatically by inserting, deleting, or reorganizing nodes
- Implementation strategies differ by type of tree
  - For CS 20 – learn to implement 2 types of *binary* trees: heaps, and binary search trees

ADTs – depends on tree type

- e.g., Heap
  - Limited operations – one insert, one remove
    - But these are very efficient
  - Mostly used to implement priority queues
    - Also can be used to sort – basis of HeapSort algorithm
- e.g., Binary Search Tree
  - More flexible remove operation (usually) – any item
  - Also flexible traverse operations – various orders
  - But no duplicate items allowed in tree – i.e., is a set
  - Main advantage is quick searching – hence the name

Are plenty of tree applications

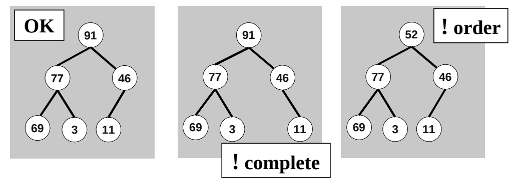
- Organizing files – directory structures are trees
- Storing strategies for computer game-players
  - What can happen if …?
    - Given each of those outcomes, what can happen next?
      - And so on, …
- Representing decision trees in general
  - Binary tree branches usually if-yes … and if-no …
- Another way to represent expressions
  - Also binary trees – internal nodes are operations, leaves are operands
- And many more

Binary trees

- Each node can have *0, 1, or 2 children only*
- i.e., a binary tree node is a subtree that is either empty, or has left and right subtrees
  - Notice this is a recursive definition
  - Concept: a leaf's "children" are two empty subtrees
- Half (+1) of all nodes in full binary tree are leaves
  - All nodes except leaves have 2 non-empty subtrees
  - Exactly 2^k nodes at each depth `k`, $\forall$`k <` (leaf level)
- A complete binary tree satisfies two conditions
  - Is full except for leaf level
  - All leaves are stored as far to the *left* as possible

Heaps

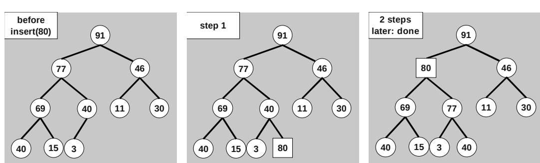
- Complete binary trees, whose items must be comparable and stored in heap order
  - Heap order – if a Max-Heap, a node's information is never less than the information of one of its children (opposite for Min-Heap)



Inserting an item in a heap

- `insertHeap` algorithm keeps complete / in order:

```
put item in first available slot; /*keep complete*/
while (new info > parent info) /*assuming Max-Heap*/
    swap info with parent;          /* "reheapify" */
```



Implementing a heap

- Convenient to implement as an array
  - Root: [1]; root children: [2,3]; their children: [4:7] …
  - Works because of binary completeness requirement – tree is full at all depths except leaves
- e.g., insertHeap algorithm
  - Step 1: `put item at end of array;`
    - `O(1)` complexity, unless array is filled up
  - Step 2 until done: `reheapify by array indexing;`
    - Have parent of `array[i]` at `array[i/2]`, $\forall$ `i>1`
    - `O(log n)` complexity to reheapify this way
- So complexity of `insertHeap` is `O(log n)` overall

Using a heap as a priority queue

- To remove highest priority item from heap:

```
remove root; /* O(1) complexity */
heapify in reverse; /* O(log n) complexity */
```

  - So overall complexity is `O(log n)`
- Meaning `O(log n)` for both insert and delete
- Compare to other priority queue strategies
  - Sorted list: insert – O(n); remove – O(1)
  - Unsorted array: insert – O(1); remove – O(n)
- Choose heap strategy if `n` is expected to be large

Representing as linked nodes

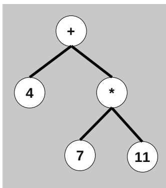
- Most trees are not as "regular" as heaps
  - So array representation wastes space, and does not accommodate changes well
- Binary tree node:

```
class TreeNode {
   Object item;  /* a data item to store in the tree */
   TreeNode left;  /* one child */
   TreeNode right;  /* other child */
}
```

  - Like lists, except each node links to *two* other nodes
- Much more flexible than array representation

Traversing trees

- Example: an expression tree (a type of "parse tree" built by advanced recursion techniques) representing this infix expression: `4 + 7 * 11`



- Infix is in-order traversal
  - Left subtree → node → right subtree
- But can traverse in other orders
  - Pre-order: node → left → right, gives prefix notation: `+ 4 * 7 11`
  - Post-order: left → right → node, gives postfix notation: `4 7 11 * +`

Binary tree traversals

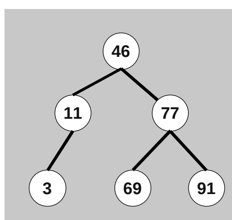
- Naturally recursive functions
  - Order of recursive calls determines traversal order
    - Remember recursive ruler tick-mark drawing?
- e.g., method to "visit" nodes in-order:

```
void inOrderTraverse(TreeNode n) {
    if (n != NULL) {
        inOrderTraverse(n.left);   /* A */
        visit(n);                  /* B */
        inOrderTraverse(n.right);  /* C */
    }
}
```

- Pre-order: B A C; Post-order: A C B

Binary search trees – BSTs

- Order rule for BSTs – say tree node is `n`:
  - Info in left subtree of `n` is less than info in `n`
  - Info in right subtree of `n` is greater than info in `n`
- Tree may not contain any duplicate info, and items must be comparable
- No rule for tree shape (except must be binary)



Searching a BST iteratively

- e.g., return reference to node with "key" item:

```
TreeNode n = root;     /* start at root node */
while(n != null && n.item != key)
/* iterate until no more branches or item is found */
    if (key < n.item)  /* search left subtree */
        n = n.left;
    else               /* search right subtree */
        n = n.right;
return n;      /* either null, or node with key info */
```

Searching a BST recursively

- External method (i.e., not a TreeNode method):

```
TreeNode findNode(Comparable key, TreeNode n)
 { if (n is null || n.item equals key)
      return n; /* works for both base cases */
   else if (key is less than n.item)
      return findNode(key, n.left);
   else return findNode(key, n.right);
 }
```

- Same complexity as iterative version
  - Notice: each iteration eliminates ½ remaining nodes
  - Similar result applies to many binary tree operations