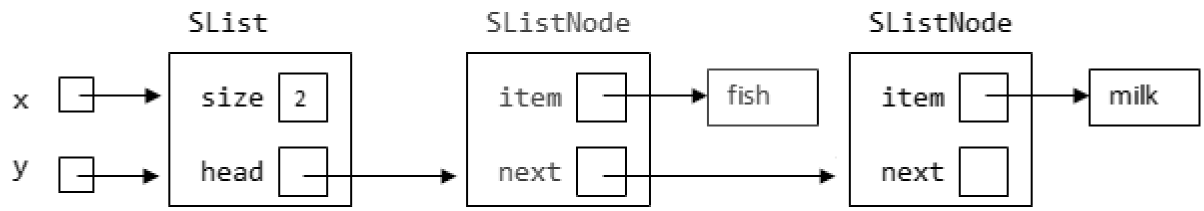
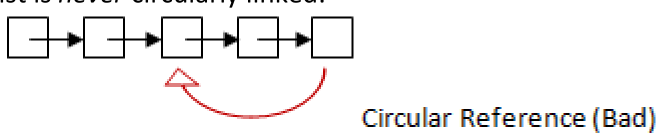
# CS 61B Data Structures

**Singly-Linked Lists** | Useful if number of items in list is not fixed. Lecture 6.




*Invariants*

- `SList`'s `size` field is always correct (representing number of `SListNode`s in the list)
- List is *never* circularly linked:




- No `next` pointer points to head (head is $1^{st}$ element or is `null`)
- Either `tail == null` or `tail.next == null`

*Performance*

- Insert and remove operations take constant time
- Find operations take linear time

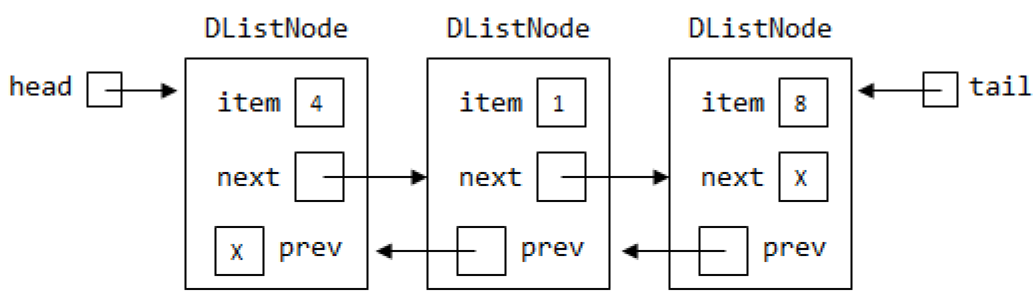
*Advantages (over Array-Based Lists)*

- Inserting item into middle of linked lists take constant time if you have reference to previous node
- Lists can keep growing until memory runs out (you'd have to reallocate new arrays otherwise)

*Disadvantages*

- Finding $n^{th}$ item of linked list takes linear time

**Doubly-Linked Lists** | Implementation of linked lists which allow for easy insertion at end of list. Lecture 7.




Note: `head` and `tail` are part of `DList`.

*Invariants*

- `size` field always represents number of `DListNode`s in list (excludes the head and `tail`)
- List is *never* circularly-linked
- Either `head == null` or `head.prev == null`; either `tail == null` or `tail.next == null`

*Performance*

- Insert and remove operations take constant time
- Find operations take linear time (although you can start from either end)

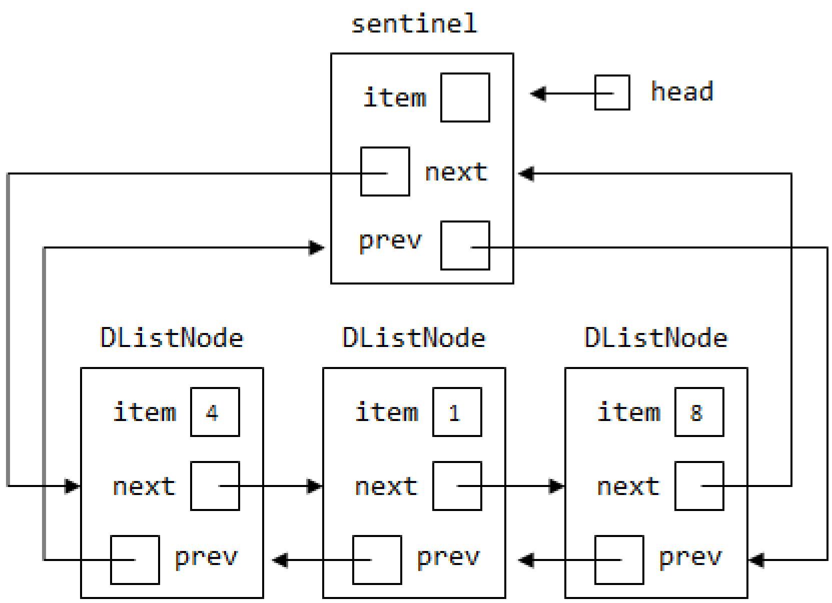
*Advantages*

- Inserting item into middle of linked lists take constant time if you have reference to previous node
- Lists can keep growing until memory runs out (you'd have to reallocate new arrays otherwise)
- Can walk through list from either end of list (due to `head` and `tail` references)

*Disadvantages*

- Requires special cases for `DList` with no items and `DList` with one item
- Finding $n^{th}$ item takes linear time (though you can start from either end)

**Circularly-Linked Lists** | Implementation of Doubly-Linked Lists which require fewer special cases. Lecture 7.




Note: head is part of DList.

*Invariants*

- For every `DList d`, `d.head != null`.
- For every `DListNode x`, `x.next != null`.
- For every `DListNode x`, `x.prev != null`.
- For every `DListNode x`, if `x.next ==`y then `y.prev != x`.
- For every `DListNode x`, if `x.prev == z`, then `z.next == x`
- A `DList's size` variable is number of `DListNodes`, not counting sentinel
- Empty `DList`: `sentinel's prev` and `next` fields point to itself

*Performance*

- Insert and remove operations take constant time
- Find operations take linear time

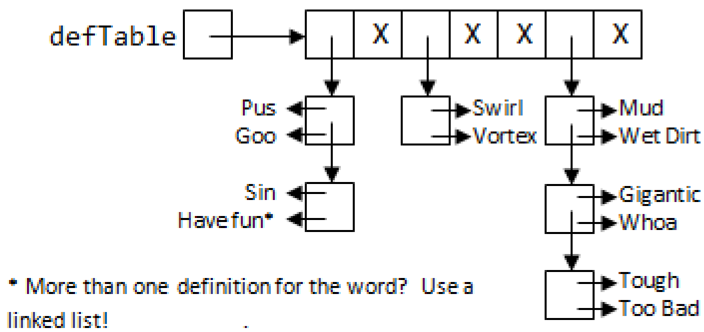
*Advantages*

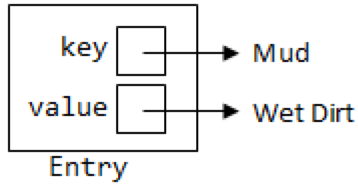
- Does not require special cases for `DList` with no items and `DList` with one item (due to sentinel node)
- Inserting item into middle of linked lists take constant time
- Lists can keep growing until memory runs out (you'd have to reallocate new arrays otherwise)
- Can walk through list from either end of list (due to sentinel node)

*Disadvantages*

- Requires special cases for `DList` with no items and `DList` with one item
- Finding $n^{th}$ item takes linear time (though you can start from either end)

**Dictionaries / Hash Tables** | Maps arbitrary key to a single value. Key must exist in dictionary. Lectures 22-23.




*Properties*

- If keys are prioritized or ordered, consider a **priority queue/binary heap** if you only want to retrieve minimum key. Use **binary search tree** if you want to retrieve any ordered key. Hash tables canNOT deal with inexact matches!
- Implemented as an array of linked lists. The array represents the buckets and the linked lists take care of collisions from the compression function.
- Good compression function: `h(hashCode) = ((a * hashCode + b) mod p) mod N`
    - `a, b` are positive integers. This has the effect of converting n into a base a number. Example: $abcd_3 = (((a * 3 + b) * 3 + c) * 3 + d)$
    - p is large, positive prime number. a and p should have no common factors.
    - N does not need to prime. p takes care of randomizing the hashcode.

*Key Methods*

(i) `public Entry insert(key, value)`
- Compute key's hash code
- Compress it to determine which bucket to look inside
- Insert entry into bucket's linked list

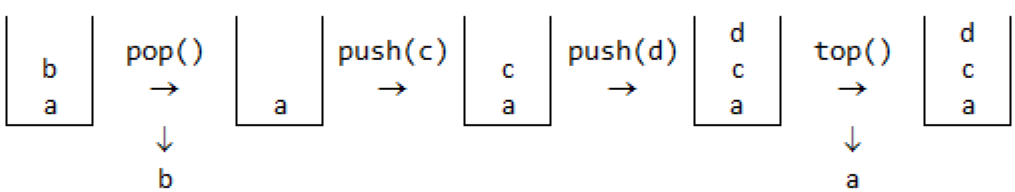
(ii) `public Entry find(key)`
- Hash the key (in same way as `insert()`) then compress
- Search list for entry with matching key
- If found, return entry; otherwise return `null`

(iii) `public Entry remove(key)`
- Hash key, search list
- Remove entry from list if found
- Return `entry` or `null`

*Performance*

- Ideally, if Load Factor $\frac{n}{N} \approx 1$, runtime is $O(1)$. ($n$ is number of keys and $N$ is number of buckets, where $n < N$)
- If Load Factor becomes too large, runtime is $O(n)$. (Dominated by linked list performance)

**Stacks** | Last-In, First-Out (LIFO). Crippled list which only manipulates element at top of stack. Lecture 23.



*Key Methods* – Stacks can be implemented as a singly-linked list with the following behaviors:

(i) `public int size();`
- Return `size` field of Singly-Linked List

(ii) `public boolean isEmpty();`
- Check if `size` field of singly-linked list is 0. If so, return true, otherwise return false.

(iii) `public void push(Object item);`
- Implement an `insertFront()` method, as done with singly-linked lists.

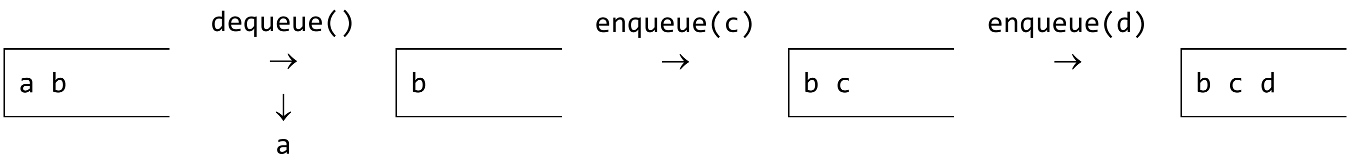
(iv) `public Object pop();`
- Implement a `removeFront()` method, as done with singly-linked lists.

(v) `public Object top();`
- Implement `front()` as in a singly-linked list, which returns the `head` of the list.

*Performance*

- All of the above methods run in $O(1)$ time.

**Queues** | First-In, First-Out (FIFO). Crippled list: read/remove from front and add to end of queue. Lecture 23.




*Key Methods* – Stacks can be implemented as a singly-linked list with the following behaviors:

(i) `public int size();`
- Return `size` field of Singly-Linked List

(ii) `public boolean isEmpty();`
- Check if `size` field of singly-linked list is 0. If so, return true, otherwise return false.

(iii) `public void enqueue(Object item);`
- Implement an `insertFront()` method, as done with singly-linked lists.

(iv) `public Object pop();`
- Implement a `removeEnd()` method, as done with singly-linked lists. This is fast if we include a `tail` pointer.

(v) `public Object top();`
- Implement `front()` as in a singly-linked list, which returns the `head` of the list.

*Performance*

- All of the above methods run in $O(1)$ time, assuming that a `tail` pointer is maintained.

**Deques** | A Double-Ended Queue where items can be inserted and removed from both ends. Lecture 23.

Implemented as a doubly-linked list with `removeFront()` and `removeBack()` methods. The idea is similar to that above for stacks and queues.

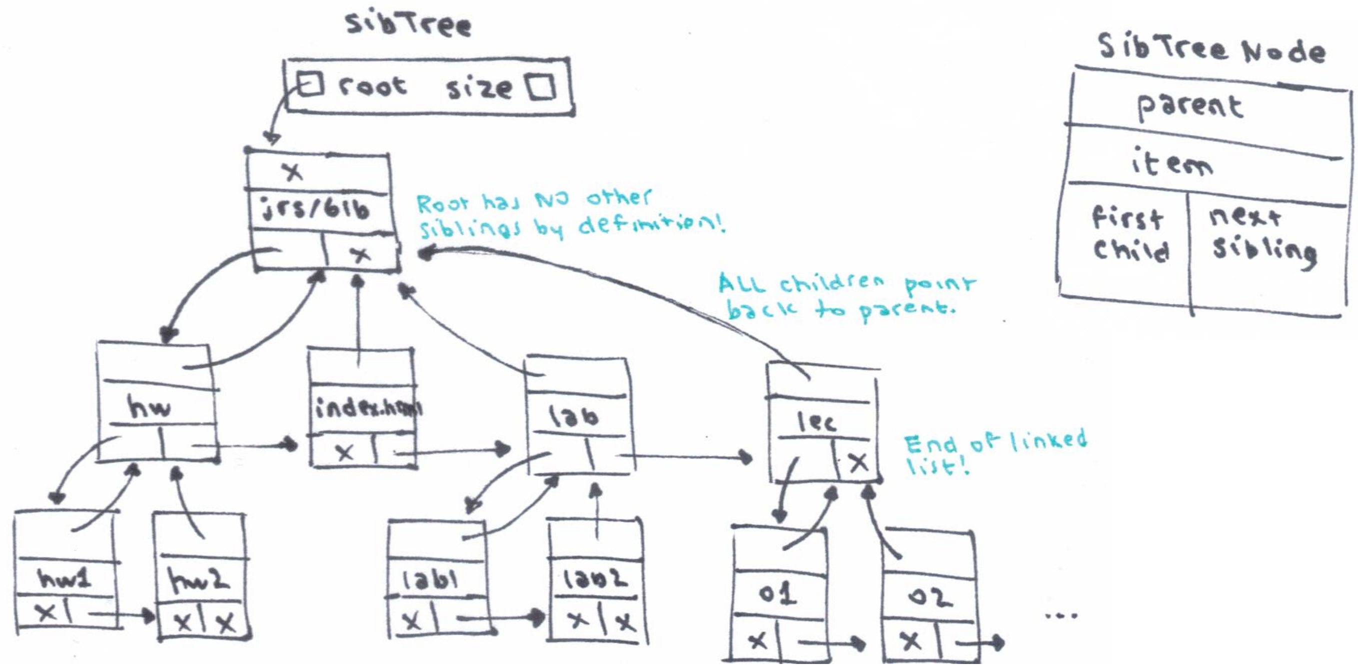
**Rooted Trees** | Set of nodes and edges that connect pairs of nodes (specialized graph). Lecture 24.

*Properties*

- Exactly one path between any two nodes of tree (as opposed to graphs, which allow any number of paths)
- Each node has a *single* parent except for root node, which has no parent.
- Root has no siblings by definition.
- Any node can have any number of children. (Binary trees, 2-3-4 trees, etc. will have different limitations)

*Implementation*

- Children are stored as a (singly) linked list: parent's `firstChild` field points to its left-most child, and each node points to its `nextSibling`.
- Each child points back to its own `parent`.




*Tree Traversal Performance*

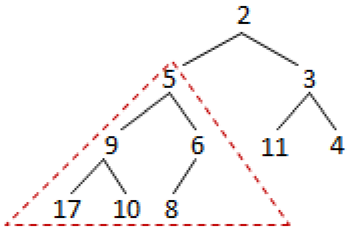
Example Tree:

```
      +
     / \
    *   ^
   / \ / \
  3  7 4  2
```

| Type | Preorder | Postorder | Inorder (Binary Trees Only!) | Level Order (by depth) |
|---|---|---|---|---|
| **Example** | Prefix Notation<br>+ * 3 7 ^ 4 2 | Postfix Notation<br>3 7 * 4 2 ^ + | Infix Notation<br>3 * 7 + 4 ^ 2 | N/A<br>+ * ^ 3 7 4 2 |
| **Order** | 1<br>/ \\<br>2 6<br>/ \| \\ / \\<br>3 4 5 7 8 | 8<br>/ \\<br>4 7<br>/ \| \\ / \\<br>1 2 3 5 6 | 4<br>/ \\<br>2 6<br>/ \\ / \\<br>1 3 5 7 | 1<br>/ \\<br>2 3<br>/ \| \\ / \\<br>4 5 6 7 8 |
| **Algorithm** | Visit current node, then visit children recursively.<br><br>Similar to depth-first search in graphs. | Visit each node's children recursively, then visit itself.<br><br>Similar to depth-first search in graphs. | Recursive function. Visit left child, then visit itself, then visit right child. | Use a queue containing only root. Dequeue node, visit it, and encode its children until queue is empty.<br><br>Similar to breadth-first search in graphs. |
| **Runtime** | Each node in tree is visited once, so runtime for all of these traversals is $O(n)$. | | | |

**Priority Queues / Binary Heap** | Maps prioritized entries to values. Limited to operations which identify or remove minimum (or some other order) key. Lecture 25.

If you have arbitrary keys which are not prioritized, consider using a **dictionary/hash table**. If you need to be able operate on any arbitrary prioritized entry in an ordered dictionary, use a **binary search tree**.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| X | 2 | 5 | 3 | 9 | 6 | 11 | 4 | 17 | 10 | 8 |

Note: Red-outlined subtree is also a binary heap.

*Properties/Invariants*

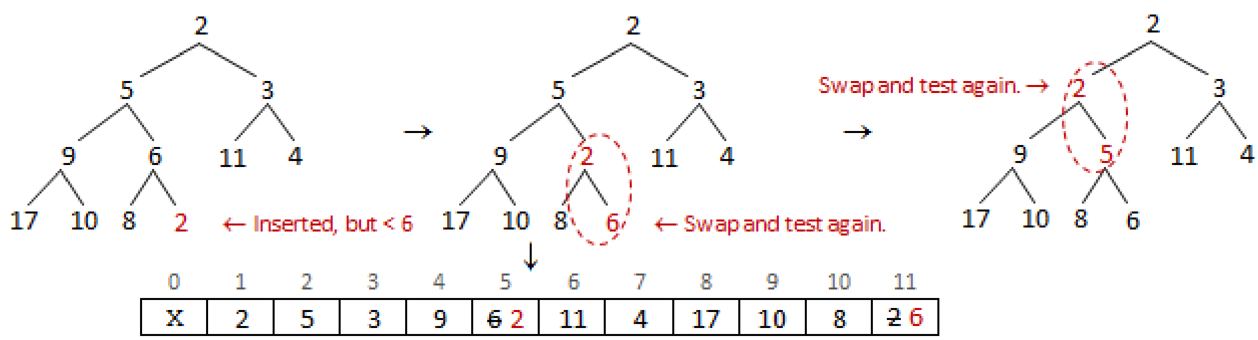
- Must satisfy <u>heap-order property:</u> no child has key less than parent's key
  - Minimum key always at top of heap! (root of tree)
- Binary heaps are complete trees – every subtree is also a complete binary tree/each row of tree is filled as much as possible
- Can be represented as either (i) a binary tree, or (ii) an array: root is in index 1, and for each node at index `i`, children are located at `2i` and `2i+1` while parent node is at $\lfloor i/2 \rfloor$.
- Keys in array are stored with level-order traversal (possible because tree is complete)

*Key Methods*

(i) `public Entry min();`

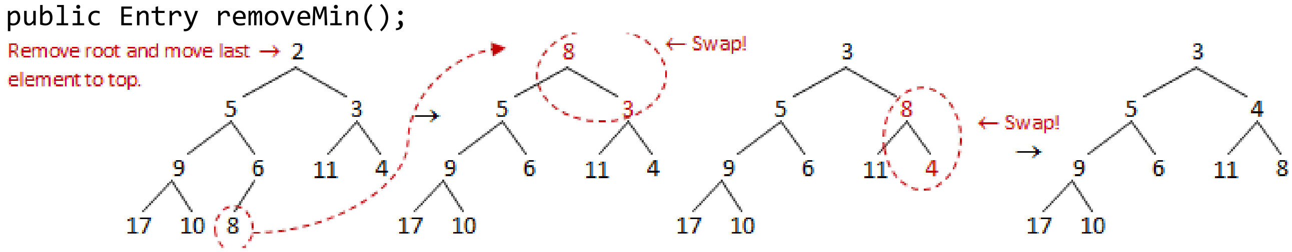
- If heap is empty, return null or throw exception
- Otherwise, return Entry at root node

(ii) `public Entry insert(Object k, Object v);`




- Insert k at first open space (bottom of tree)
- Bubble k up tree – swap with parent key while k < parent key

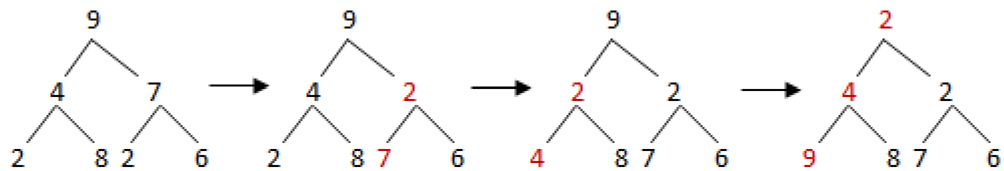
(iii) `public Entry removeMin();`




- Remove entry at root and save for return value
- Fill hole with last entry in tree x to ensure tree is complete

- Bubble x down tree by swapping x with minimum child key until x ≤ child key OR it becomes leaf

(iv) `public void bottomUpHeap();`



- Randomly dump all keys into a complete tree. (Algorithm reorders keys as necessary.)
- Begin at last internal (non-leaf) node. If heap-order property is violated (child key is less than this key), swap with minimum child key. Subtree now satisfies heap-order property.
- Continue moving backwards until root is reached.

*Performance*

| | Binary Heaps | Sorted List/Array | Unsorted List/Array |
|---|---|---|---|
| `min()` | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| `insert()` | | | |
| Worst Case: | $\Theta(\log n)$ | $\Theta(n)$ | $\Theta(1)$ |
| Best Case: | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ |
| `removeMin()` | | | |
| Worst Case: | $\Theta(\log n)$ | $\Theta(1)$ | $\Theta(n)$ |
| Best Case: | $\Theta(1)$ | $\Theta(1)$ | $\Theta(n)$ |
| `bottomUpHeap()` | $\Theta(n)$ – G&T p.371 | -- | -- |

**Ordered Dictionary / Binary Search Trees** | Implementation of dictionary with ordered keys. Generalizes functionality of priority queues and dictionaries. Lecture 26.

If your keys are not ordered and inexact key matches are not required, use a **dictionary/hash table**. If your keys are prioritized but you only need to retrieve one type of key (say, the minimum), consider using a **priority queue/binary heap**.

```
         18
        /  \
     12      25
    / \     / \
   4  15   25  30
  /   / \      /
 1  13  17   28
  \   \        \
   3  14        29
```

*Properties*

- Each node can have up to two children, with each child designated either a left child or right child
- All nodes except root have parent reference

*Invariants*

- For any node X, every key in left subtree of X ≤ X's key.
- For any node X, every key in right subtree of X ≥ X's key.
- Inorder traversal of a binary search tree visits nodes in sorted order.

*Key Methods (G&T p.446)*

(i) `public Entry find(Object k);`

- Start at root and repeatedly compare key k with current key.
  - If k == key, return the corresponding Entry.
  - If k < key, move to left child and repeat.
  - If k > key, move to right child and repeat.
  - Base cases: (i) node has been found, or (ii) node has no more children in wanted direction.
- To find approximate matches, store largest key less than key *and* smallest key greater than key.

(ii) `public Entry first(); public Entry last();`

- If tree is empty, return null.

- `first()` returns the minimum key, which is the left-most node. Start at root and repeatedly go to left child until you reach node with no left child.
- `last()` returns maximum key, which is right-most node. Start at root and repeatedly go to right child until you reach node with no right child.

(iii) `public Entry insert(Object k, Object v);`

- Follow same path that `find()` follows. When you reach null reference, replace null with new node with entry (k, v).

```
        18                            18
       /  \                          /  \
     12     25                     12     25
    / \     / \                   / \     / \
   4  15  25  30  -insert(2)->   4  15  25  30
  /   /  \    /                 /   /  \    /
 1  13  17  28                 1  13  17  28
  \   \       \                 \   \       \
   3  14      29                 3  14      29
                                /
                               2
```

(iv) `public Entry remove(Object k);`

- Find key k in tree, or return null if it is not in tree

| Chd | Algorithm |
|---|---|
| 0 | • Detach child from its parent and return. |
| 1 | • Move n's child up to take n's place. n's child becomes the child of n's parent. (Redraw the pointer to bypass the removed node.) |

```
        18                           18
       /  \                         /  \
     12     25                    12     25
    / \     / \                  / \     / \
   4  15  25  30 -remove(30)->   4  15  25  28
  /   /  \    /                 /   /  \      \
 1  13  17  28                 1  13  17      29
  \   \       \                 \   \
   3  14      29                 3  14
  /                             /
 2                             2
```

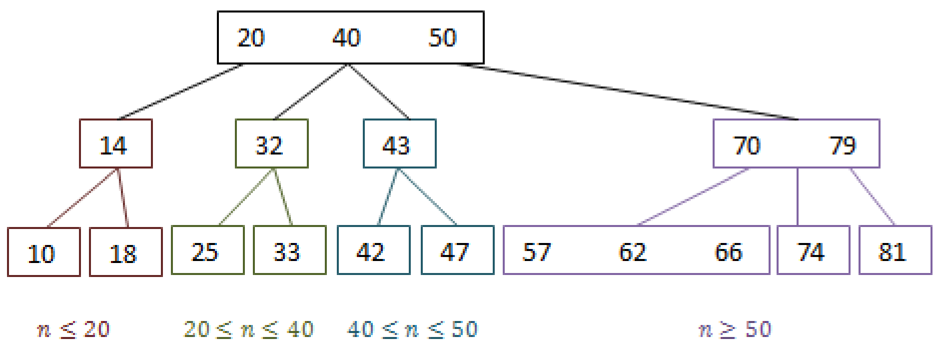
| Chd | Algorithm |
|---|---|
| 2 | • Let x be node in n's right subtree with smallest key (as if we were calling `first()` on that subtree). • Remove x and replace n's entry with x's entry |

```
        18                           18                          18
       /  \                      +--/  \                        /  \
     12     25                   |12|    25                   13     25
    / \     / \                  /-\+   / \                  / \     / \
   4  15  25  28  -remove(12)->  4  15  25  28       ->     4  15  25  28
  /   /  \      \               /+-/+ \      \             /   /  \      \
 1  13  17      29             1 |13| 17     29           1  14  17      29
  \   \                         \+-\+                      \
   3  14                         3  14                      3
  /                             /                          /
 2                             2                          2
```

*Performance*

| Best-Case Scenario | Worst-Case Scenario | General Case |
|---|---|---|
| Perfectly-Balanced Tree<br>o<br>/ \\<br>o o<br>/\\ /\\<br>o o o o<br>/\\ /\\ /\\ /\\<br>o oo oo oo o | Severely-Imbalanced Tree<br>1<br>\\<br>2<br>\\<br>3<br>\\<br>4<br>\\<br>5<br>\\<br>6 | Any tree that satisfies binary search tree invariants |
| Runtime: $O(\log n)$<br>Tree has maximum depth $\log_2 n$, which is maximum number of recursive calls.<br>**2-3-4 Trees** are always perfectly-balanced. | Runtime: $O(n)$ | Runtime: $O(\log n)$ |

**2-3-4 Trees** | Tree structure that is always perfectly balanced, so runtime is always $O(\log n)$ time. Lecture 27.




**2-3-4 Trees** are always perfectly balanced, but insert and remove operations are more complicated and generally take longer (despite same asymptotic performance). If there's no need for a perfectly-balanced tree, consider using **Binary Search Trees**.

*Properties*

- Perfectly-balanced tree: there are between $2^h$ and $4^h$ leaves and $n \geq 2^{h+1} - 1$.
- Each node (except leaves) have between 2-4 children.
- Each node contains 1-3 keys. $\#\ of\ children = \#\ of\ keys + 1\ OR\ 0$. Subtree keys are stored as above.
- Inorder traversal can be defined on 2-3-4 tree to return keys in sorted order.

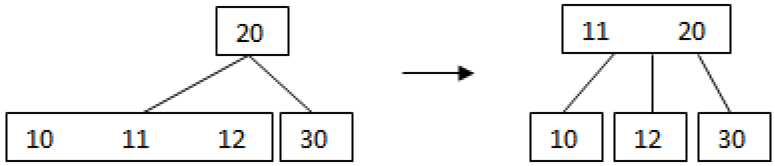
*Key Methods*

(i) `public Entry find(Object k);`

- Similar to binary tree: start at root and check k against keys. Move to appropriate child until k is found or a leaf is reached.

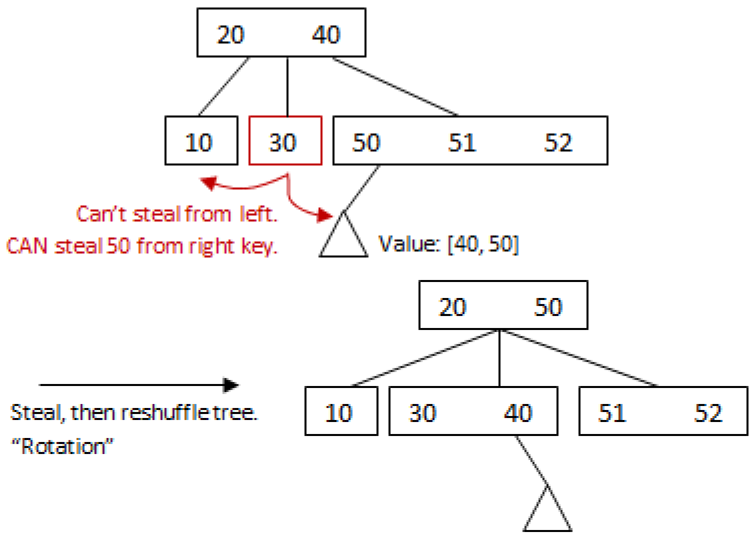
(ii) `public Entry insert(Object k, Object e);`

- Walks down tree in search of k.
- If it finds k, it proceeds to k's "left child" (child immediately to left of k) and continues.
- Whenever `insert()` encounters 3-key node, middle key is moved up to parent node. (Always break up 3-key nodes on path!).

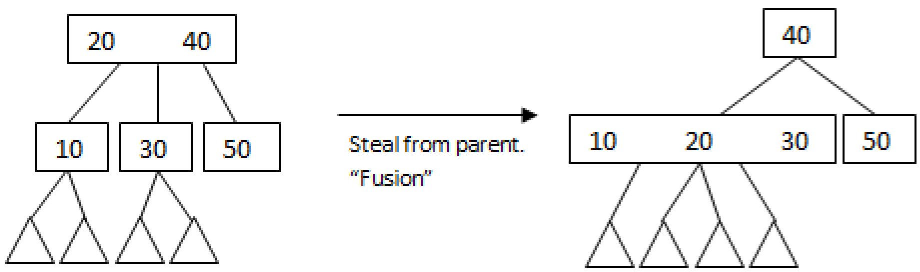



(iii) `public Entry remove(Object k);`

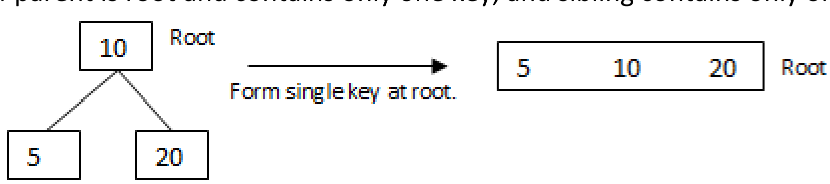
- Find key k. If it's in a leaf, remove it. If it's in an internal node, replace it with entry with next higher key (which is always in a leaf). (Same method as binary search tree)
- Get rid of 1-key nodes on path by restructuring tree:
  - (i) If adjacent siblings have more than 1 key: Rotation – try to steal key from adjacent sibling.




  - (ii) If adjacent siblings only have 1 key: Fusion – try to steal key from parent.




  - (iii) If parent is root and contains only one key, and sibling contains only one key:




*Performance*

- Each tree contains $2^h$ to $4^h$ leaves, so number of entries $n \geq 2^{h+1} - 1$. Height $h \in O(\log n)$.
- Constant time per node (but by larger factor than binary search tree)
- Number of nodes proportional to height of tree. All operations are $O(h) = O(\log n)$.

**Graphs |** Any set of *V* of vertices and *E* of edges which connect the vertices together. Lecture 28.

*Properties*

- Can be directed or undirected. If undirected, $(v, w) = (w, v)$.

- Maximum of one copy of each edge. If directed graph, $(v, w) \neq (w, v)$!

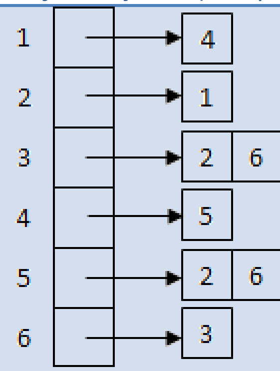
*Representations and Performance*

**Adjacency Matrix** (for complete graphs)

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | - | - | - | T | - | - |
| 2 | T | - | - | - | - | - |
| 3 | - | T | - | - | - | T |
| 4 | - | - | - | - | T | - |
| 5 | - | T | - | - | - | T |
| 6 | - | - | T | - | - | - |

- Table of booleans, where true indicates that edge exists.
- If undirected graph, adjacency matrix is symmetric.
- If weighted graph, use matrix of ints (weights).
- Lookup Runtime: $O(1)$
- DFS, BFS Runtime: $O(|v^2|)$
- Memory Use: $O(|v^2|)$ (total possible edges)

**Adjacency List** (for sparse graphs)




- Dictionary or array of lists
- Lookup Runtime: $O(1)$
- DFS, BFS Runtime: $O(|v| + |e|)$
- Memory Use: $O(|v| + |e|)$ (number of vertices and edges from each vertex).

*Algorithms*

- Kruskal's Algorithm – Minimum Spanning Trees
  - (i) Create new graph T with same vertices as G with no edges.
  - (ii) Make list of all edges in G.
  - (iii) Sort edges by weight, from lowest to highest.
  - (iv) Iterate through edges in sorted order. If u and w are not connected, add (u, w) to T.

  Runtime Performance: $O(|v| + |e| \log|v|)$
- Depth-First Search (DFS) – similar to preorder traversal in trees. Search as deeply as possible. Code in Lecture 28.
  - (i) Start at arbitrary vertex and visit the vertex. Mark vertex as visited.
  - (ii) Iterate and recursively run `dfs()` on each edge for each vertex that has not been visited.

  Alternatively, use a stack!
- Breadth-First Search (BFS) – similar to level-order traversal in trees. Search by distance. Code in Lecture 29.
  - (i) Start at arbitrary vertex, mark as visited, and enqueue it.
  - (ii) Dequeue a vertex and visit it. Pass the origin into `visit()` as parameter.
  - (iii) Enqueue each edge connected to an unvisited vertex.
  - (iv) Repeat until queue is empty.