CS61B – Data Structures

Anmol Parande

Spring 2019 - Professor Josh Hug

Contents

1 Lists

1.1 SLLists

A Single-Linked-List is a one way Linked List (i.e each node keeps a reference to its successor)

```
class SLListNode<T> {
    T item;
    SLListNode next;
}
```

To make the logic of methods in SLList easier, we often make the first node in an SLList a **Sentinel Node**. The sentinel node contains a null item. If the SLList is empty, then sentinel.next = null. Otherwise, sentinel.next is the first item of the SLList.

1.1.1 Big-Θ Bounds

Operation	Time Complexity
Size	$\Theta(1)$
AddFront	$\Theta(1)$
AddLast	$\Theta(N)$
DeleteFront	$\Theta(1)$
DeleteLast	$\Theta(N)$
Get	$\Theta(N)$

1.2 DLLists

Since SLLists take long time to add to the front and to the back, a Double Linked List (DLList) can speed these up.

```
class DLListNode<T> {
    T item;
    DLListNode next;
    DLListNode prev;
}
```

Like in SLLists, we can have a sentinel node to help us simplify operations. sentinel.next will be the first element in the list whereas sentinel.prev is the last element of the list.

1.2.1 Big-Θ Bounds

Operation	Time Complexity
Size	$\Theta(1)$
AddFront	$\Theta(1)$
AddLast	$\Theta(1)$
DeleteFront	$\Theta(1)$
DeleteLast	$\Theta(1)$
Get	$\Theta(N)$

1.3 ArrayList

LinkedLists are a poor choice for retrieving information inside of them unless we are only looking for the front or back element (or if the list is short). For longer lists, we can make an array-based list called an ArrayList.

```
public class ArrayList<T> {
    T[] items
    public void resize(int N);
}
```

An ArrayList simply maintains an array of all the items it has. The caveat is that when the array becomes full, we must resize it. The way we resize the array affects its runtimes. If we increase the size of the array by a multiplicative factor (i.e doubling the array each time), then the amortized (on average) runtime will still be constant.

1.3.1 Big-Θ Bounds

Operation	Time Complexity
Size	$\Theta(1)$
AddFront	$\Theta(1)$
AddLast	$\Theta(1)$
DeleteFront	$\Theta(1)$
DeleteLast	$\Theta(1)$
Get	$\Theta(1)$

2 Trees

2.1 Binary Search Tree

In a Binary Search Tree, each node has a value, a left child, and a right child.

```
public class Node<T> {
    T value;
    Node left;
    Node right;
}
```

2.1.1 Invariants

- Every key in the left subtree is less than the root value
- Every key in the right subtree is greater than the root value
- No duplicates are in the BST
- The minimum element is the left most node
- The maximum element is the right most node

2.1.2 Big-Θ Bounds

Operation	Worst Case	Best Case
Height	$\Theta(N)$	$\Theta(logN)$
Search	$\Theta(N)$	$\Theta(logN)$
Insert	$\Theta(N)$	$\Theta(logN)$
Delete	$\Theta(N)$	$\Theta(logN)$

The worst case runtime for a BST is when the tree is "spindly," meaning there are more nodes on either the right or left side (i.e, it is not "balanced")

2.1.3 Hibbard Deletion

Hibbard Deletion is a special method of efficient deletion. You can either

> Find the smallest key larger than the root value (called the successor) and make it root

OR

> Find the largest key smaller than the root value (called the precessor) and make it root

2.2 2-3, 2-3-4 Tree

A 2-3-4 tree is the same idea as a BST, but it has extra operations to make it balanced. Nodes in 2-3-4 tree's can be "stuffed", meaning they have more than 1 value inside them. If a node is overstuffed, then it gives an item to the parent and splits the range of the children.

2.2.1 Invariants

- All leaves are the same distance from the source
- Non-leaf with k items has exactly k+1 children
- The same invariants as BSTs

2.2.2 Big-Θ Bounds

Operation	Worst Case	Best Case
Height	$\Theta(logN)$	$\Theta(logN)$
Search	$\Theta(logN)$	$\Theta(logN)$
Insert	$\Theta(logN)$	$\Theta(logN)$

2.3 Left Leaning Red Black Tree

An LLRB is a BST but stays balanced using tree rotations. Each LLRB has a unique 2-3 tree because the red links represent stuffed nodes. They are guaranteed to be balanced.

2.3.1 Tree Rotations

Rotate Left

```
y = x.right
x.right = y.left
y.left = x
```

Rotate Right

```
y = x.left
x.left = y.right
y.right = x
```

2.3.2 Inserting into a LLRB

- When inserting, use a red link
- If there are two consecutive red links, rotate the tree right
- If there is a right leaning 3-node, rotate the tree left
- If a node has two red children, color flip

2.3.3 Invariants

- No node has two red links
- Every path to a leaf has the same number of black links
- There is a one-one correspondance to a 2-3 tree

2.3.4 Big-Θ Bounds

Operation	Worst Case	Best Case
Height	$\Theta(logN)$	$\Theta(logN)$
Search	$\Theta(logN)$	$\Theta(logN)$
Insert	$\Theta(logN)$	$\Theta(logN)$

2.4 KD-Tree

A KD-Tree is a multidimensional BTree. Each subsequent level splits on a different dimension of the data. KD-Tree's are useful for finding the nearest point in the data to a given set of points.

2.4.1 Invariants

- Each new level partitions data by a new dimension (nodes on the right are larger than root for that particular feature, vice versa for nodes on the left)

3 Hashtables

A Hashtable is simply an array of Linked Lists. Each node in the Linked list contains a value.

3.1 Invariants

- Equal objects have equal hashcodes
- Objects which can change should not be stored

3.2 Big-Θ Bounds

Operation	Worst Case	Best Case	Caveats
Contains	$\Theta(N)$	$\Theta(N)$	No Resize
Add	$\Theta(N)$	$\Theta(N)$	No Resize
Contains	$\Theta(N)$	$\Theta(1)$	On Average
Add	$\Theta(N)$	$\Theta(1)$	On Average

The best case runtime occurs when we have a good hashcode (items are evenly distributed among buckets). The worst case runtime occurs when we have a bad hashcode (items all fall into one bucket).

4 Heaps

A heap is made similar to a BST but with different invariants. They are useful in implementing priority queues.

4.1 Invariants

- Complete: All nodes are as left as possible. Nodes are only missing from the bottom level
- Each node's value is less than the value of the children (in a min-heap, for a max-heap, it's the opposite)

4.2 Big-Θ Bounds

Operation	Worst Case	Best Case
Add	$\Theta(logN)$	$\Theta(1)$
Get Smallest	$\Theta(1)$	$\Theta(1)$
Remove Smallest	$\Theta(logN)$	$\Theta(logN)$

Note: These runtimes are amortized if the heap is represented as an array. The true worst case runtime is $\theta(N)$ because of the resize operation.

5 Tries

A trie are like HashMaps except the nodes are Strings. They make prefixing operations very easy.

5.1 Invariants

- Each node stores a single letter
- Nodes can be shared by many keys
- Nodes ending in keys are marked

5.2 Big-Θ Bounds

Operation	Worst Case	Best Case
Get	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$

6 Disjoint Sets

A disjoint set has two operations, *connect* and *isConnected*. Two items are connected if they are connected to the same item.

6.1 QuickFind

In QuickFind, the representation of the data is an array whose elements are the index of the group to which the object belongs.

6.1.1 Big-Θ Bounds

Operation	Worst Case	Best Case
Constructor	$\Theta(N)$	$\Theta(N)$
connect	$\Theta(N)$	$\Theta(N)$
isConnected	$\Theta(1)$	$\Theta(1)$

6.2 Quick Union

In Quick Union, the representation of the data is an array whose elements are the parent of the item.

6.2.1 Big-Θ Bounds

Operation	Worst Case	Best Case
Constructor	$\Theta(N)$	$\Theta(N)$
connect	$\Theta(N)$	$\Theta(1)$
isConnected	$\Theta(N)$	$\Theta(1)$

6.3 Weighted Quick Union

In Weighted Quick Union, the elements of the array are the parent of the item. The difference from normal Quick Union is that when connecting two trees, the root of the smaller tree is linked to the root of the larger tree. The root node will hold the number of children (the weight) as its element.

6.3.1 Big-Θ Bounds

Operation	Worst Case	Best Case
Constructor	$\Theta(N)$	$\Theta(N)$
connect	$\Theta(logN)$	$\Theta(1)$
isConnected	$\Theta(logN)$	$\Theta(logN)$

7 Graphs

Representation	addEdge	print	hasEdge	Space
Adjacency Matrix	$\Theta(1)$	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V^2)$
List of Edges	$\Theta(1)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
Adjacency List	$\Theta(1)$	$\Theta(V+E)$	$\Theta(ElogV)$	$\Theta(E+V)$