10: multiway trees

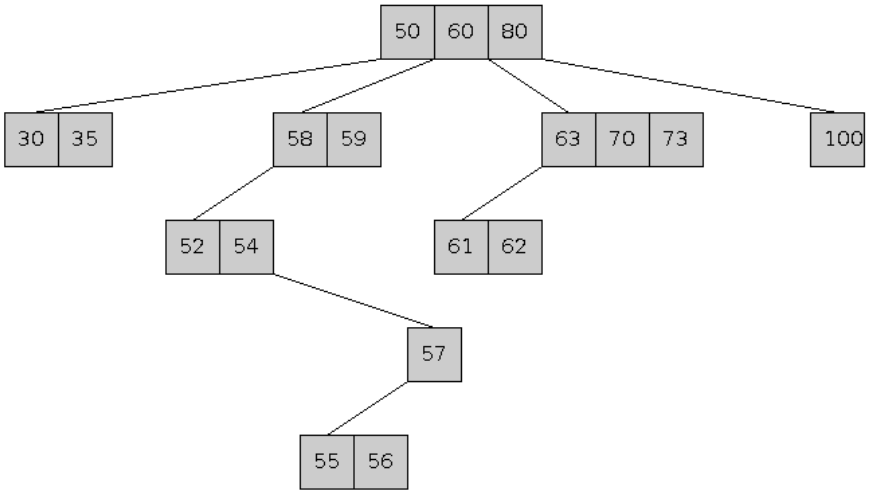
B-Trees | B*-Trees | Prefix B+-Trees | Bitmap Index | 2-3-4 Trees | Red Black Trees | C++ Sets and Maps

Multiway Search Trees

Properties of a multiway search tree

1. Each node has *m* children and *m*-1 keys.
2. The keys in each node are in ascending order
3. The keys in the first *i*-1 children are smaller than the *i*th key.
4. The keys in the last *m*-i-1 children are larger than the *i*th key.

Example of a multi-way tree:



B-Trees

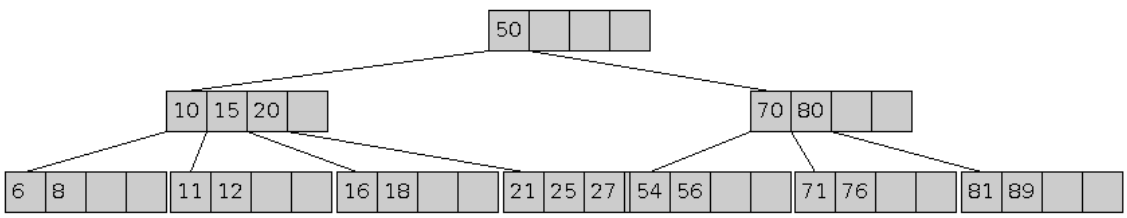
A B-Tree is a multiway search tree that is designed to keep trees on hard disks. Hard disks arranged into (usually) 512-byte blocks and keeping one key per block is slow.

- access_time = seek_time + rotational_delay + transfer_time
- access_time = 40ms + 10ms + 5ms = 55ms
- Increasing the amount of data read by a little has a small effect on the time to do a random disk read.

A *B-Tree of order m* is a multiway search tree with the following properties:

1. The root has at least two subtress unless it is a leaf
2. Each nonroot and each nonleaf node holds *k*-1 keys and *k* pointers to subtrees where ceiling(*m*/2)<=*k*<=*m*
3. Each leaf node holds *k*-1 keys where ceiling(*m*/2)<=*k*<=*m*
4. All leaves are on the same level

A B-Tree of order 5:



The order of a B-Tree in a database is determined by:

- order = block_size / key_size

Searching in a B-Tree

Searching in a B-Tree is fairly straightforward:

```
1:  // btreeSearch.cpp - download here
2:
3:  BTreeNode * btreeSearch(int key, BTreeNode * curr){
4:    if(curr != NULL){
5:      int i;
6:      //seek the the closest key
7:      for(i = 0; i < curr->sizeKeys() && curr->getKey(i) < key; ++i);
8:
9:      if(i >= curr->sizeKeys() || curr->getKey(i) > key){
10:       //if the above for loop broke because the key was not found, recurse
11:       return btreeSearch(key, curr->getPointer(i));
12:     } else {
13:       //else we found the node
14:       return curr;
15:     }
16:   } else {
17:     return NULL;
18:   }
19: }
```

B-Tree Search Time Complexity

Worst Case Searching:

- Each node has the smallest allowable number of pointers (q = ceiling(m/2)). (The number of keys is q-1)
- The number of keys per level follows a geometric progression
  - Root: 1 key
  - Next level: (q-1) + (q-1) keys = 2(q-1) keys
  - Next level: 2q(q-1) keys
  - Next level: 2q^2(q-1) keys
  - Leaves level: 2q^(h-1)*(q-1) keys
- $1 + \sum_{i=0}^{h-2} (2q^i)$ * (q - 1) keys in the B-Tree
- $\sum_{i=0}^{n} (q^i)$ = (q^(n+1) - 1) / (q - 1) (formula for the sum of the first n elements in a geometric progression)
- num_keys >= $1 + 2*(q-1) * \sum_{i=0}^{h-2} (q^i)$ = 1 + 2 * (q - 1) * (q^(h-1) - 1) / (q - 1)
- num_keys >= -1 + 2q^(h-1)
- Solving for h.
- h <= log_q((n + 1)/2) + 1

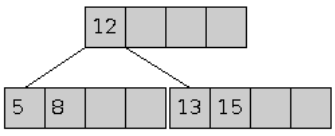
For m = 200, n = 2,000,000, h <= 4. For 2 million keys in a database, you need only four disk seeks. If you keep the root block in ram, you need only three.

Insertion in a B-Tree

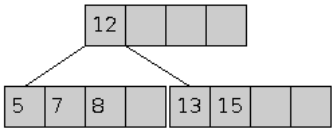
Given an incoming key, we go to the leaf and place it there, if there is room. When the leaf is full, another leaf is created, keys are divided amoung leaves and one key is promoted to the parent. If the parent is full, the process is repeated.

There are three cases when inserting into a B-Tree

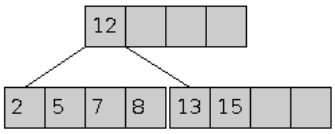
1. A key is placed in a leaf that still has room:
    1. Start:



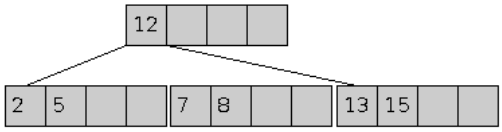
    2. Insert 7. (Needed to shift 8 over)



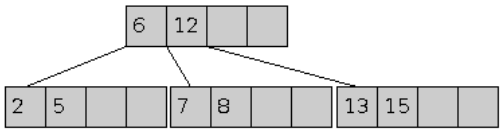
2. A key is to be placed in a leaf that is full:
    1. Start: (Inserting 6)



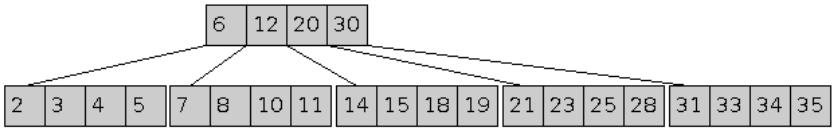
    2. Create a new node by splitting the node where the insertion should occur.



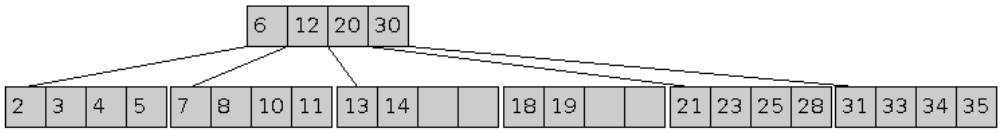
    3. Place key at parent and shift other keys and pointers



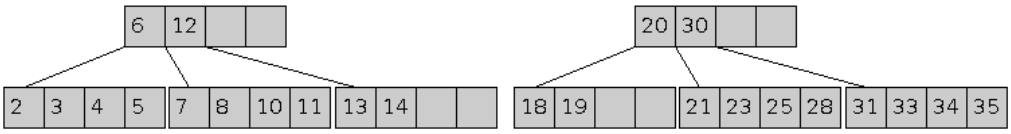
3. A key is to be placed in tree that is full:
    1. Start: (Inserting 13)



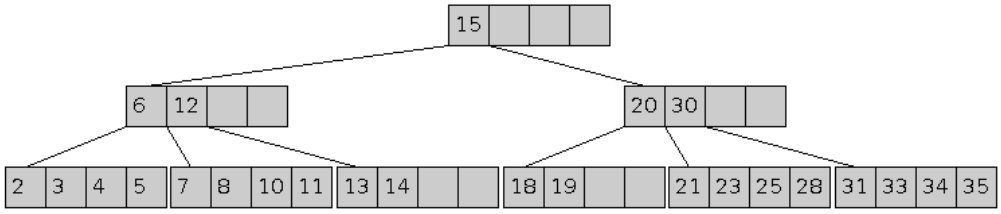
    2. Create a new node by splitting the node where the insertion should occur. Save off the 15.



    3. The 15 needs to go in the parent, but there is no room. Split the parent.



    4. Insert the 15 above the two new nodes created.



Algorithm:

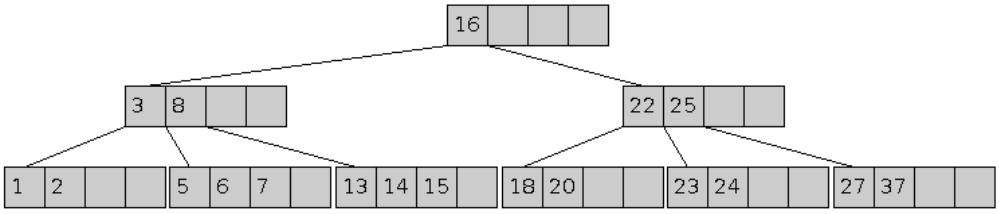
```
1:  algorithm btreeInsert(root, key)
2:    find a leaf node to insert key;
3:    while(true)
```

```
4:      find a proper position in array for key;
5:      if node is not full
6:        insert key in and shift other elements over;
7:        return;
8:      else
9:        split node into node1 and node2;
10:       distribute keys and pointers evenly between node1 and node2;
11:       if node was the root
12:         create a new root as parent of node1 and node2;
13:         put the middle key from the split node in the root and set the pointers;
14:         return;
15:       else
16:         node = node's parent;
```

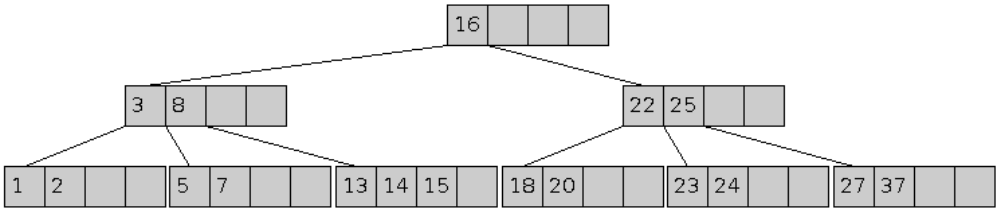
Deletion of a Key from a B-Tree

Deletion is like insertion, but with more cases. Care must be taken to ensure that nodes are never less than half full.

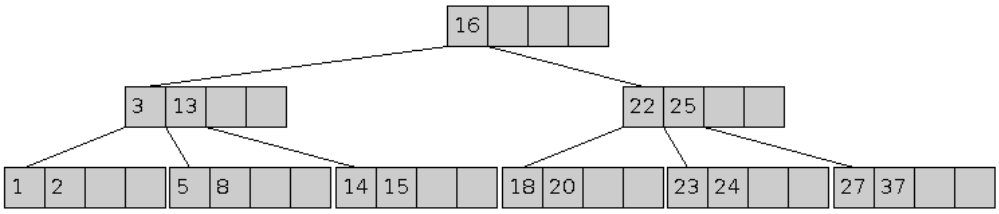
1. Start:



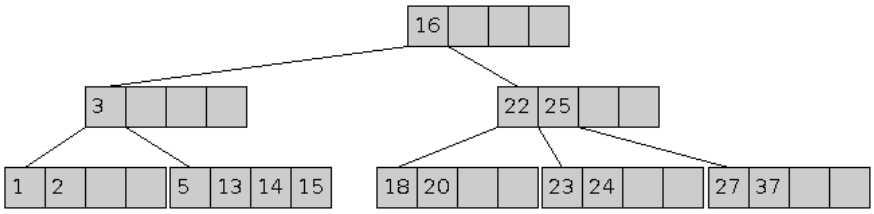
2. Delete 6. This is a simple remove and shift case in a leaf.



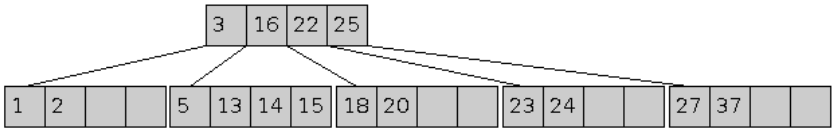
3. Delete 7. The number of keys in the leaf is less than half. Take the 8 from the parent and shift the 13 from the next sibling. This is okay because after the removal of 13, the next sibling is still half full



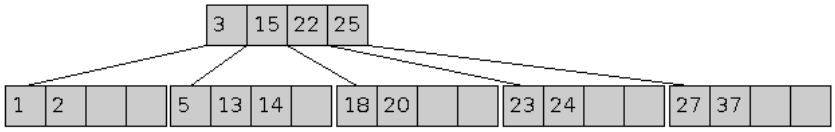
4. Delete 8.
    1. After removing 8, the leaf underflows. Take the 13 from the parent and the 14 and 15 from the next sibling to fill the leaf.



    2. After doing this, the parent is less than half full. A special case here is that we will be filling a root node that has only one key. Create a new root from the parent, it's sibling and the old root. The lowest levels in this diagram are kept the same.



5. Delete 16. Here we are deleting from a non-leaf. The key to be deleted is replaced with it's immediate predecessor (which can only be found in a leaf). The problem is then reduced to deleting a key from the leaf.




```
1:  algorithm btreeDelete(root, key)
2:    node = BTreeSearch(key, root);
3:    if(node == null)
4:      return;
5:    if(node is not a leaf)
6:      find a leaf with the closest predecessor S of key;
7:      copy S over key in node;
8:      node = the leaf containing S;
9:      remove S from node;
10:   else
11:     delete key from node;
12:   while(true)
13:     if node does not underflow
14:       return;
15:     else if there is a sibling of node with enough keys
16:       redistribute keys between node and its sibling;
17:       return;
18:     else if node's parent is the root
19:       if the parent has only one key
20:         merge node, its sibling, and the parent to form a new root;
21:       else
22:         merge node and its sibling;
23:       return;
24:     else
25:       merge node and its sibling;
26:       node = its parent;
```

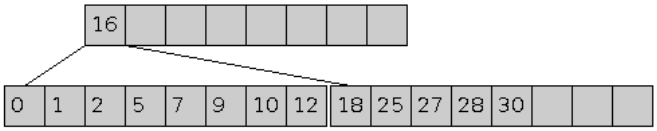
B*-Trees

B*-Trees are a varient of B-Trees that was introduced by Donald Knuth and Douglas Comer. In B*-Trees, all nodes (except the root) are required to be 2/3rds full, rather than half.

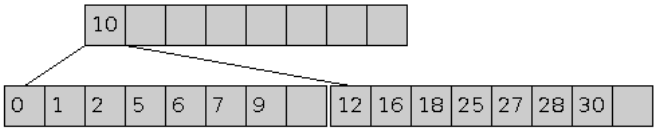
- The frequency of node splitting is decreased by delaying a split. When a node is full, the keys in the full node are evenly distributed with a sibling.
- A split occurs when two siblings are full and the nodes are split into three.
- The average utilization of B*-Trees was found to be 84% while B-Trees are approximately 69%

Splits in B*-Trees are delayed by attempting to redistribute keys among the node, its parent and its sibling

1. Start:

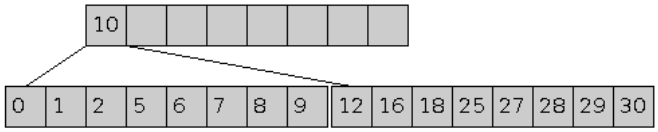


2. Insert 6 and delay split by redistributing keys

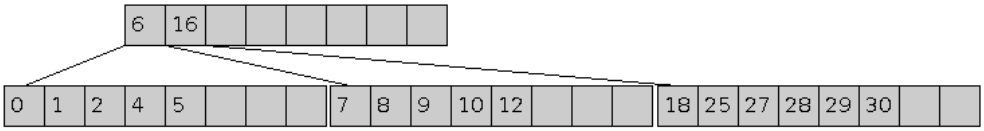


If a node and its sibling are both full, a split is done and keys are distributed among three nodes

1. Start:



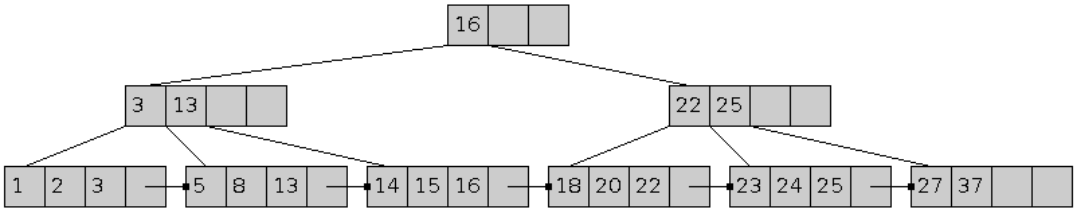
2. Insert 4. Split and redistribute among three nodes



B+ Trees

In B-Trees and B*-Trees, it is simple to write code to do an in-order traversal. However, if the blocks represent disk blocks, only one element from an internal node is accessed before switching to another disk block. This can be costly (time).

In B+ Trees, the leaves contain all the keys in the tree (not just the leave keys) and the leaves have links forming a linked list.



Insertion and deletion in B+ Trees is similar to B and B* Trees.

Notes from Wikipedia [1]

Systems using B+ Trees:

1. Filesystems:
    1. NTFS
    2. NSS
    3. XFS
    4. JFS
2. Relational Databases:
    1. IBM DB2
    2. Informix
    3. Microsoft SQL Server
    4. Oracle 8
    5. Sybase ASE
    6. SQLite
3. Key-value Databases:
    1. CouchDB
    2. Tokyo Cabinet

The keys in a node of B-Trees, B*-Trees and B+ Trees can be stored as Binary Search Trees instead of an array to speed deleting and adding elements [1]

Bitmap Index

Table:

Row	Algorithm	Worst Case Time Complexity
0	bubblesort	O(n^2)
1	heapsort	O(nlgn)
2	insertion sort	O(n^2)
3	mergesort	O(nlgn)

4	quicksort	O(n^2)
5	selection sort	O(n^2)
6	smooth sort	O(n^2)

Bitmap index:

Row	0	1	2	3	4	5	6
O(nlgn)	0	1	0	1	0	0	0
O(n^2)	1	0	1	0	1	1	1

The index is kept in compressed format using a bitvector

2-3-4 Trees [2]

A **2-3-4 search tree** is a tree that either is empty or has three types of nodes:

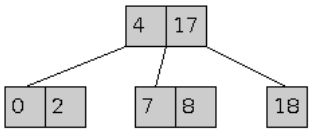
1. **2-nodes**: with one key and two pointers
2. **3-nodes**: with two keys and three pointers
3. **4-nodes**: with three keys and four pointers

A **balanced 2-3-4 search tree** is a 2-3-4 search tree with all links to empty trees at the same distance from the root.

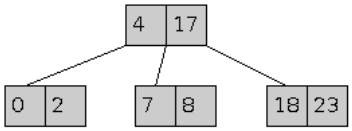
Insertion

There are three primary cases for insertion into 2-3-4 Trees:

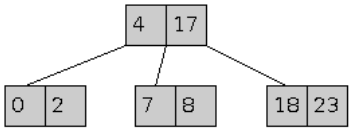
1. Search Terminates on a 2-node:
    1. Start:



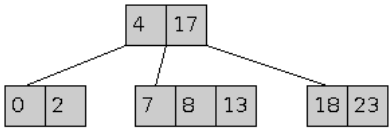
    2. Insert 23. Make 2-node into a 3-node



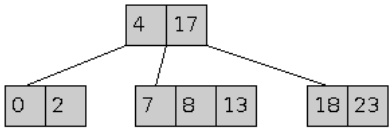
2. Search Terminates on a 3-node:
    1. Start:



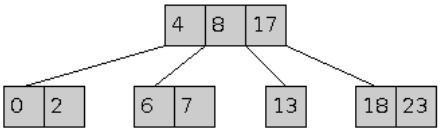
    2. Insert 13. Make 3-node into a 4-node



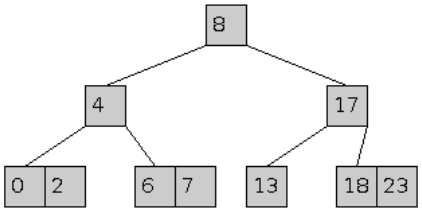
3. Search Terminates on a 4-node:
    1. Start:



    2. Insert 6. Split the 4 node according to the B-Tree rules.



3. Special Case: The root has become a 4-node. Split it into a triangle of 2-nodes. This is the only case where the tree height grows by one.



2-3-4 Tree Time Complexities

- **Search:** O(log n)
- **Insert:** O(log n)
- **Delete:** O(log n)

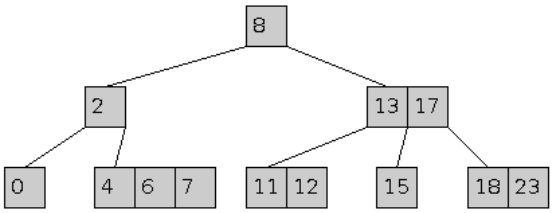
Red-Black Trees [1]

2-3-4 Trees provide O(log n) performance for insertion, delete and search. However they can be hard to implement.

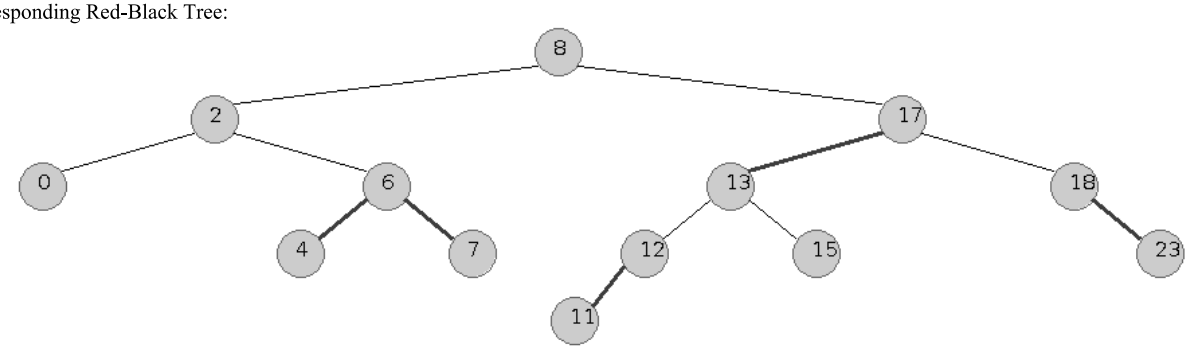
Red-Black Trees use standard 2-nodes but add one extra bit of information. Links can be either red or black.

- **2-nodes**: represented by one 2-node connected by a single **black link**
- **3-nodes**: represented by two 2-nodes connected by a single **red link**
- **4-nodes**: represented by three 2-nodes connected by **red links**

2-3-4 Tree:



Corresponding Red-Black Tree:



Requirements of Red-Black Trees [2]

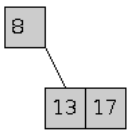
1. The root is black
2. Two red nodes cannot exist in a row

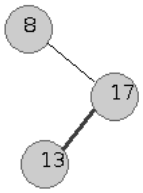
Red-Black Tree Strengths:

- Search takes no additional time over a regular binary tree search. The color is never examined and there is no rebalancing on search
- Insertion only needs rebalancing when we see 4-nodes

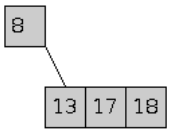
Insertion

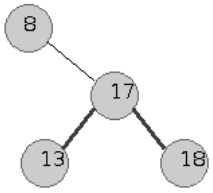
1. Inserting into a 2-node with a 1-node as a parent
    1. Start:



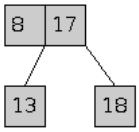


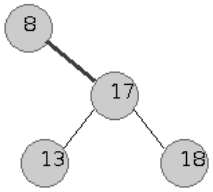
    2. Insert 18



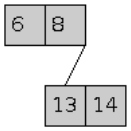


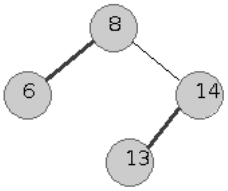
    3. Push the red links up to the parent



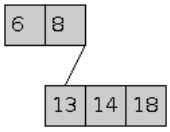


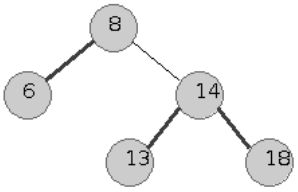
2. Simple insertion into a 2-node with a 2-node as a parent
    1. Start:



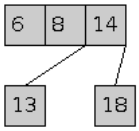


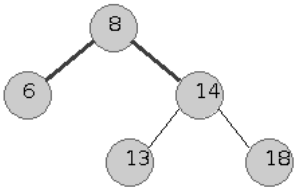
    2. Insert 18



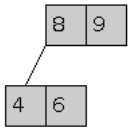


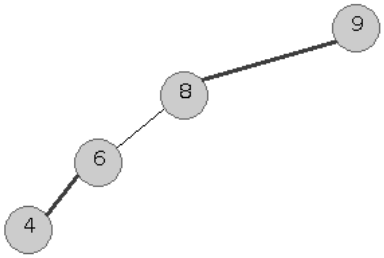
    3. Push the red links up to the parent



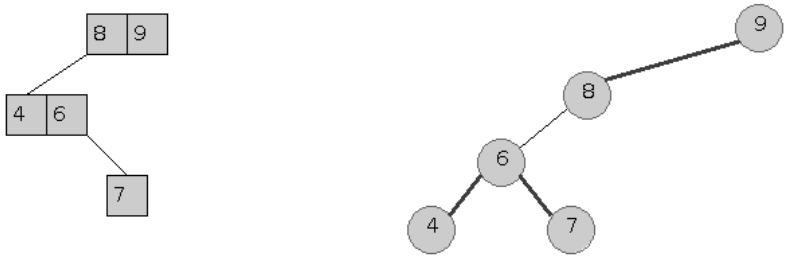


3. Single Rotation insertion into a 2-node with a 2-node as a parent
    1. Start:

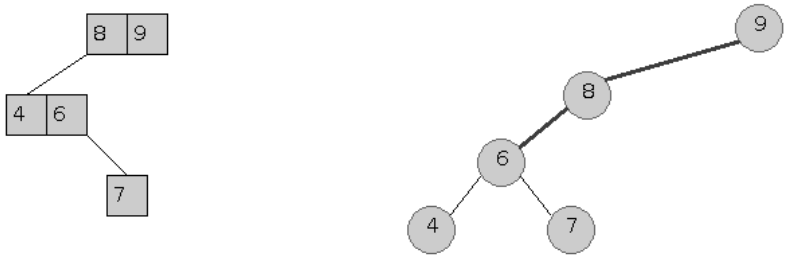




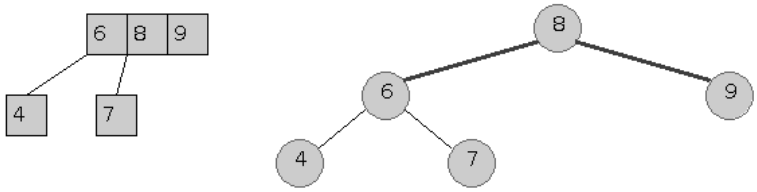
    2. Insert 7



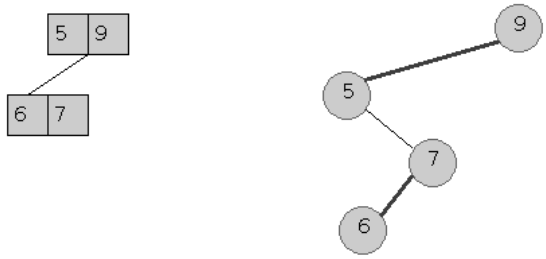
3. Push the red links up to the parent



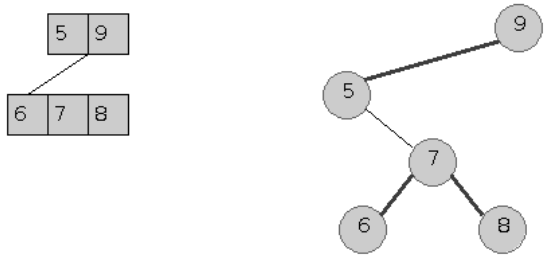
4. Two red links are in a row. Both links are left links, do a single right rotation about 8.



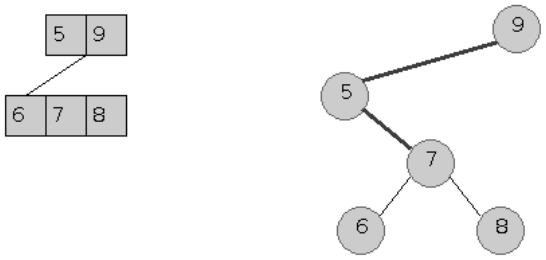
4. Double Rotation insertion into a 2-node with a 2-node as a parent
    1. Start:



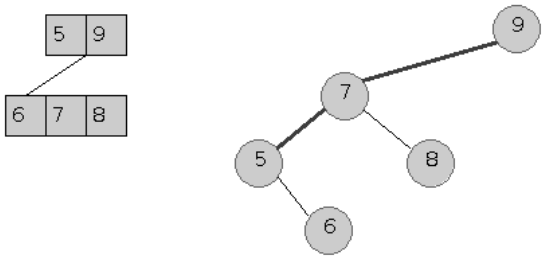
    2. Insert 8



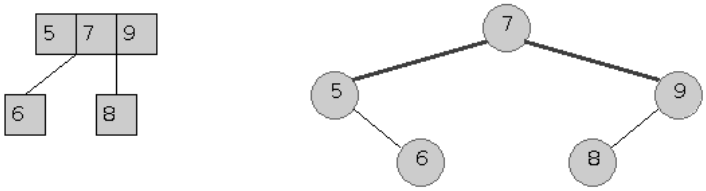
    3. Push the red links up to the parent



4. Two red links are in a row. One link is left and another is right. First do a left rotation about 5.



5. Then do a right rotation around 9.



C++ Sets, Multisets

Sets are implemented as Binary Search Trees. Sets cannot have duplicate keys. Multisets can have duplicate keys.

Set Code:

```
1:  // cppSetExample.cpp - download here
2:
3:  #include <iostream>
4:  #include <set>
5:
6:  int main(int argc, char * argv[]){
7:
8:    std::set<int> set1;
9:    set1.insert(20);
10:   set1.insert(10);
11:   set1.insert(50);
12:   set1.insert(23);
13:   set1.insert(20);
14:
15:   std::set<int>::iterator iter;
16:   std::cout << "items in set1: " << std::endl;
17:   for(iter = set1.begin(); iter != set1.end(); ++iter){
18:     std::cout << "   " << *iter << std::endl;
19:   }
20:   //prints:
21:   //  items in set1:
22:   //    10
23:   //    20
24:   //    23
25:   //    50
26:
27:   return 0;
28: }
```

Multiset Code:

```
1:  // cppMultisetExample.cpp - download here
2:
3:  #include <iostream>
4:  #include <set>
5:
6:  int main(int argc, char * argv[]){
7:
8:    std::multiset<int> mset1;
9:    mset1.insert(20);
10:   mset1.insert(10);
11:   mset1.insert(50);
12:   mset1.insert(23);
13:   mset1.insert(20);
14:
15:   std::multiset<int>::iterator iter;
16:   std::cout << "items in mset1: " << std::endl;
17:   for(iter = mset1.begin(); iter != mset1.end(); ++iter){
18:     std::cout << "  " << *iter << std::endl;
19:   }
20:   //prints:
21:   //  items in mset1:
22:   //    10
23:   //    20
24:   //    20
25:   //    23
26:   //    50
27:
28:   return 0;
29: }
```

C++ Maps, Multimaps

Maps are Binary Search Trees with keys that also hold on to values. Maps cannot have duplicate keys, Multimaps can.

Map Code:

```
1:  // cppMapExample.cpp - download here
2:
3:  #include <iostream>
4:  #include <map>
5:  #include <string>
6:
7:  int main(int argc, char *argv[]){
8:
9:    std::map<std::string, std::string> dictionary;
10:   dictionary["array"] = "a systematic arrangement of objects, usually in rows and columns";
11:   dictionary["vector"] = "an automatically resizable array";
12:   dictionary["queue"] = "a container where the next element is given in FIFO order";
13:   dictionary["stack"] = "a container where the next element is given in LIFO order";
14:
15:   std::cout << "an array is: " << dictionary["array"] << std::endl;
16:   std::cout << std::endl;
17:
18:   std::cout << "the dictionary contents: " << std::endl;
19:   std::map<std::string, std::string>::iterator iter;
20:   for(iter = dictionary.begin(); iter != dictionary.end(); ++iter){
21:     std::pair<std::string, std::string> item = *iter;
22:     std::cout << item.first << ": " << item.second << std::endl;
23:   }
24:   std::cout << std::endl;
25:
26:   dictionary.erase("array");
27:   std::cout << "after erasing 'array', ";
28:   if(dictionary.find("array") != dictionary.end()){
29:     std::cout << "dictionary contains array" << std::endl;
30:   } else {
31:     std::cout << "dictionary does not contain array" << std::endl;
32:   }
33:
34:   //prints:
35:   //  an array is: a systematic arrangement of objects, usually in rows and columns
36:   //
37:   //  the dictionary contents:
38:   //  array: a systematic arrangement of objects, usually in rows and columns
39:   //  queue: a container where the next element is given in FIFO order
40:   //  stack: a container where the next element is given in LIFO order
41:   //  vector: an automatically resizable array
```

```
42:   //
43:   //  after erasing 'array', dictionary does not contain array
44:
45:   return 0;
46: }
47:
```

Multimap Code:

```
1:  // cppMultimapExample.cpp - download here
2:
3:  #include <iostream>
4:  #include <map>
5:  #include <string>
6:
7:  int main(int argc, char *argv[]){
8:
9:    std::multimap<std::string, int> course_catalog;
10:   course_catalog.insert(std::pair<std::string, int>("cis", 252));
11:   course_catalog.insert(std::pair<std::string, int>("cis", 275));
12:   course_catalog.insert(std::pair<std::string, int>("cis", 300));
13:   course_catalog.insert(std::pair<std::string, int>("cse", 283));
14:   course_catalog.insert(std::pair<std::string, int>("cse", 382));
15:   course_catalog.insert(std::pair<std::string, int>("cse", 484));
16:   course_catalog.insert(std::pair<std::string, int>("ele", 231));
17:   course_catalog.insert(std::pair<std::string, int>("ele", 232));
18:   course_catalog.insert(std::pair<std::string, int>("ele", 312));
19:
20:   std::multimap<std::string, int>::iterator iter;
21:   iter = course_catalog.find("cse");
22:   std::cout << "courses in cse: " << std::endl;
23:   while(true){
24:     std::pair<std::string, int> item = *iter;
25:     if(item.first == "cse"){
26:       std::cout << item.first << item.second << std::endl;
27:       ++iter;
28:     } else {
29:       break;
30:     }
31:   }
32:
33:   //prints:
34:   //  courses in cse:
35:   //  cse283
36:   //  cse382
37:   //  cse484
38:
39:   return 0;
40: }
41:
```

References

1. http://en.wikipedia.org/wiki/B_plus_tree
2. Robert Sedgewick, "Algorithms in C++, Parts 1-4", Third Edition. ISBN: 0-201-35088-2, pages 560-573