Heap

A priority queue must at least support the following operations on a dynamic set S of elements:

- Insert(S,k): inserts a new element x into S with key k.
- Maximum(S): returns the element of S with the maximum value.
- ExtractMax(S): removes and returns the element with the maximum value.
- ChangeKey(S,x, k): changes the value of element x's key to the new value k. This value could be larger or smaller than the value it is replacing.

Heap data structure allows one to perform priority queue operations efficiently.

One can view a max-heap as a binary tree where each node and the key stored each node (indicating the priority) is greater than the keys of its child nodes. Thus the root of a max heap is the largest element of the heap. The heap is almost a complete binary tree at every level except the bottom level. The bottom level is filled from left to right. If the heap structure has height h storing n keys, $2^h \leq n \leq 2^{h+1} - 1$. Heap data structure allows all of the operations mentioned to be performed in logarithmic time.

The following figure is obtained from CLRS.

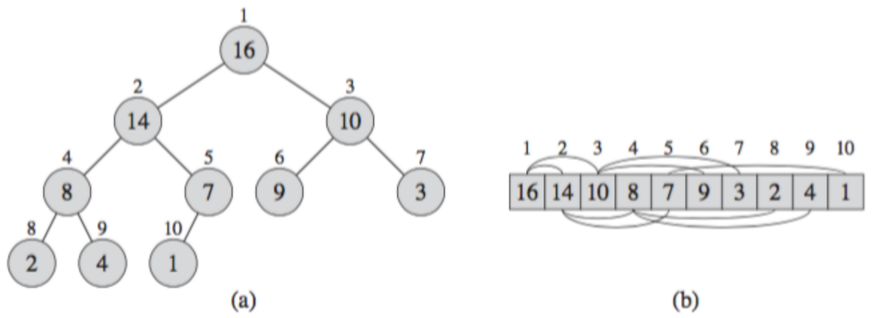


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

The heap can be implemented in an array. The root is stored at index 1, and if a node is stored at index i, its left child has index $2i$ and the right child has index$2i+1$. The parent of node i is stored at index $\lfloor \frac{i}{2} \rfloor$.

Initially, we need to build a heap of a set of keys. Surely, we can build it by adding each of the n keys one at a time requiring $O(n\log n)$ total time. However, the required heap can be constructed in $O(n)$ using a divide and conquer algorithm whose recurrence relation will look like $T(n) = 2T(\frac{n}{2}) + O(logn)$. Check that this realizes $T(n) \in O(n)$.

We can increase or decrease the value of a key at node u in $O(\log n)$ time. When the value of the key at node u is increased, the update step of the heap is to push u up the heap so that max-heap property is not violated. If the key value at node u is decreased, the update step of the heap requires node u to be pushed down.

For Dijkstra's algorithm to compute the single source shortest path tree in a graph $G = (V, E)$, min-heap can be used to maintain the information. The time complexity of Dijkstra's algorithm in that case is $O((|V| + |E|)\log|V|)$.

Sets and Dictionaries

In this section we consider a set of objects. The members of a set are drawn from a single universe: set of numbers; set of words; set of records identified by keys (viz. student records with student ids being the keys); etc. In a typical implementation of a dynamic set, each element is represented by an object whose fields can be examined and manipulated. Some kinds of dynamic sets assume that one of the objects fields is an identifying **key** field. If the keys are all different, we can think of dynamic set as being a set of key values. The object may contain **other data**. Here are some abstract operations that might be useful in applications involving sets:

Search(x, S): Return 'true' if x is an element of S, otherwise, return 'false'.

Union(S,T): Return $S \cup T$.

Intersection(S,T): Return $S \cap T$.

Difference(S,T): Return $S - T$.

MakeEmptySet(): Return the empty set Φ.

IsEmpty(S): Return 'true' if S is an empty set, else return 'false'.

Insert(x,S): Add x to S. (This has no effect if $x \in S$ already.)

Delete(x,S): Remove x from S. (This has no effect if $x \notin S$ already.)

Equal(S,T): Return 'true' if $S = T$, otherwise return 'false'.

Min(S): Return the smallest member of the set S (Elements are linearly ordered.)

Dictionary: A set of abstract data type with just operations *MakeEmptySet, IsEmptySet, Insert, Delete, and Membership* is called a **dictionary**.

0.1 Elementary Data Structures (Chapter 10; CLRS)

We can represent a dynamic set in a linear list. For each of the four types of lists in the following, the asymptotic worst case running time for each dynamic set operation is listed below. (We assume that S has n objects.)

	unsorted , singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked	sorted, array
Search(x,S)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Insert(x,S)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Delete(x,S)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
MakeEmptySet()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IsEmptySet(S)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

0.2 Binary Search Trees (Chapter 12; CLRS)

A **binary search tree (BST)** is a binary tree with the following properties:

- Each node in the BST stores a **key**, and optionally, some auxiliary information. We are assuming that all the keys are distinct.
- The key of every node in a BST is strictly greater than all the keys to the left and strictly smaller than all the keys to the right.

The **height** of a BST is the length of the longest path from the root to a leaf.

- A tree with one node has height 0.
- An empty tree has height -1, by convention.

The minimum (maximum) element of S can be found by following the left (right) child pointers from the root until a null link is encountered. The worst running time is again $O(h)$ where h is the height of the tree. Note that $0 \leq h \leq n-1$.

0.2.1 Querying a BST

A BST is shown in Figure 1.

Given a pointer to the root of the BST and a key k, the following recursive function returns the node that contains the key k.

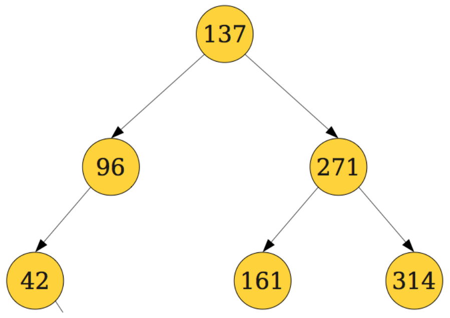


Figure 1: A binary search tree

```
=======================================
Function TreeSearch(r,k)

if r = null or k = key[r]
    then return r
if k < key[r]
    then return TreeSearch(leftChild[r],k)
    else return TreeSearch(rightChild[r],k)

========================================
```

The worst case running time is proportional to the height of the tree, say h. In the worst case, h could be as large as $n-1$. In the best case, h is $\lfloor \log_2 n \rfloor$.

0.2.2 Inserting into a BST

Insertions cause BST, representing a dynamic set, to change. We need to modify the search tree maintaing the search tree property.

Here are the steps to follow for inserting a key k.

Insert(k,S)

1. Determine the leaf node v whose left or right child will store k.
2. If $key[v] < k$, k will be stored in the left subtree.
3. if $key[v] > k$, k will be stored in the right subtree.

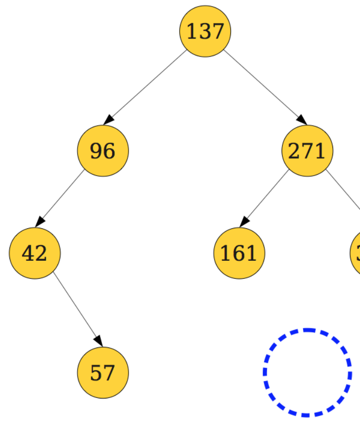


Figure 2: Determine the location in BST to insert 166

The following figures 2 and 3 depict the steps to insert 166.

Runtime Analysis

The Insert(k,S) costs $O(h)$ time. Once the leaf node location is determined, the remaining steps take $O(1)$ (i.e. constant) time. Hence $O(h)$ is the runtime cost to insert an element into a BST of height h. Note that $0 \leq h \leq n-1$.

0.2.3 Deleting a key

Deletions also cause BST to change.

Here are the steps to follow to delete a key k from the BST.

Delete(k,S)

1. Determine the node z that stores k. This is done by calling Search(k,S).
2. If z is a leaf node (Figure 4(a)), we delete the node z.
3. If z has just one child (Figure 4(b)) we delete z and set the child node of z to be the child node of the $parent(z)$. This way the BST property is maintained (why?).
4. If z has two children (Figure 4(c)), determine $Successor(z)$ node y. Replace the content of z with the content of y, and then delete node y. Note that the successor node y can have at most one child (why?).

Runtime Analysis

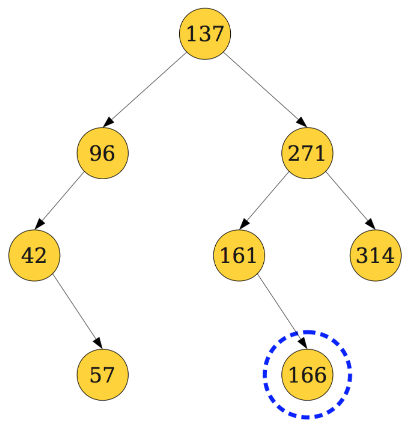


Figure 3: Insertion is complete. The updated tree is a BST.

- The time complexity is $O(h)$ since $Search(k,S)$ and $Successor(z)$ both take $O(h)$ time in the worst case. Here h is the height of BST.
- In the best case, $h \in O(\log n)$.
- In the worst case, $h \in \Theta(n)$.
- **Challenge:** Keep the height of BST, under insertion and deletion, to $\Theta(\log n)$ all the time, where n is the number of nodes in the tree.

	unsorted , singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked	sorted, array	BST, worst case	BST, best case
Search(x,S)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Insert(x,S)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Delete(x,S)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
MakeEmptySet()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IsEmptySet(S)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

1 Balanced Search Trees (2-3 Trees)

There are several important classes of problems where it is required to have the dictionary operations in dynamic settings to run in logarithmic time in the worst case.

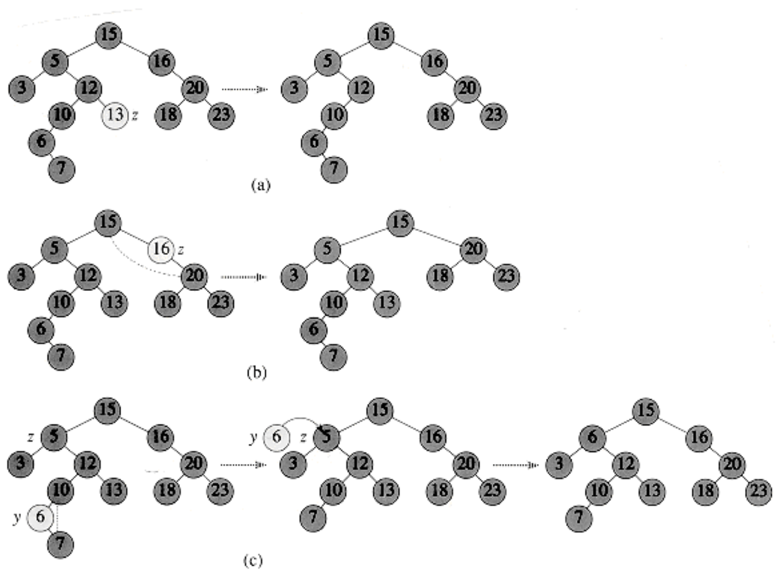


Figure 4: Deleting a node z (containing k) from a BST. In each of the following cases, the node actually removed is lightly shaded. (a) If z no child, just remove it. (b) If z has one child, we splice out z. (c) If z has two children, we splice out $y = Successor(z)$ which has at most one child, and then replace the contents of z with the contents of y.

In the following we present a data structure that will allow us to process, on-line, sequences of dictionary operations on sets. As we have observed, binary search tree, initially balanced, can become unbalanced while executing a sequence of INSERT and DELETE operations.

There are many ways to maintain a balanced tree under dictionary operations. The examples include AVL tree, 2-3 tree, splay trees, red-black tree, 2-3-4 tree, B-trees etc. In the following we discuss the method of *2-3* trees. Manipulating 2-3 trees are conceptually simple.

Definition: Our 2-3 tree is a tree with the following properties.

- The tree consisting of a single vertex is a 2-3 tree.
- Each vertex which is not a leaf has 2 or 3 children. The children are labeled first, second, third from left to right.
- Every path from the root to a leaf vertex is of the same length.

- All the elements of the set S are stored at the leaf level.

Lemma 1.1 *Let T be a 2-3 tree with height h. The number of vertices of T is between $2^{h+1}-1$ and $\frac{3^{h+1}-1}{2}$.*

Proof Consider T to be a complete binary tree of height h. In this case, the number of vertices is $2^0+2^1+\ldots+2^h$ which is $2^{h+1}-1$. Therefore, $2^{h+1}-1$ is the lower bound on the number of vertices of a 2-3 tree with height h.

The upper bound on the number of vertices is realized when the branching factor of every non-leaf vertex is three. In this case, the number of vertices of T is $3^0+3^1+\ldots+3^h$ which is $\frac{3^{h+1}-1}{2}$. ∎

The elements of S are stored at the leaves ordered by some linear order $\leq$, as follows. If element a is stored to the left of element b, the $a \leq b$ must hold. We will assume that the ordering of elements is based on one field of a record that forms the element type: this field is called the *key* field. Each record has two fields: key field, and the information field.

Interior vertices: We use the interior vertices of the tree to store information in order to facilitate the operations to be performed on the elements placed at the leaves. Every interior vertex v stores extra two pieces of information:

- **L[v]:** It is the largest element of S assigned to the subtree whose root is the first child of v.

- **M[v]:** It is the largest element of S assigned to the subtree whose root is the second child of v.

Figure 5 shows a 2-3 tree where the values of L and M are attached to the interior vertices. Both the 2-3 trees store the same elements at the leaves.

It is easy to show that

Lemma 1.2 *This implies that any 2-3 tree storing n elements at the leaves has height $O(\log n)$.*

In the following we assume that 2-3 tree T stores n elements of S at the leaves. The lointer *root* points to the root node of T.

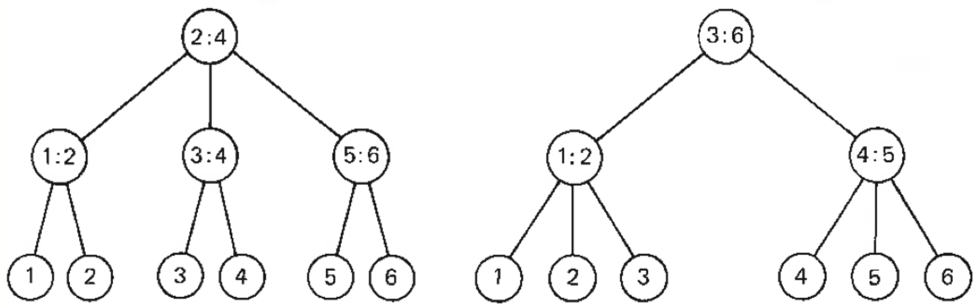


Figure 5: 2-3 Tree

1.1 Notations

We will use the following notations for the rest of discussions.

S: Set of n objects. Each object is a 2-tuple $(key, records)$.

T: Given 2-3 tree stores the elements of S at leaves.

root: The root node.

f[v :] The first child of v. If there is no child, $f(v) = null$.

s[v :] The second child of v.

t[v :] The third child of v. The third child is null if the degree of v is two.

p[v :] The parent node of v

L[v :] Largest element in the subtree rooted at $f[v]$.

M[v :] Largest element in the subtree rooted at $s[v]$.

1.2 Search(x,v)

Search the 2-3 tree T for the object with key x. The start vertex is v. Initially, v is the root node. The operation fails if no such item is found.

```
==============================================
SEARCH(x,r)

if r is a leaf then
    if key[r] = x return r
    else return null
endif

if x <= L[r] then return Search(x, f[r])
    else
       if r has two children or x <= M[r] then
           return Search(x, s[r])
           else return Search(x, t[r])
       endif
endif
===============================================
```

It is easy to show that

Lemma 1.3 *The cost of Search(x,root) is* $O(\log n)$.

1.3 Insert(x,v)

To insert a new element x into a 2-3 tree, we must locate the position for the new leaf u that will contain x. Which interior node is going to be the parent of a new leaf? This is done by trying to locate element x in the tree. The search for x terminates at a vertex z. The last interior vertex we see on the search is $p[z]$, the parent node of z. $p[z]$ should be the parent of the new leaf. We need to make sure that

- Number of children of $p[z]$ should not be more than three.
- The L[] and M[] information of the vertices along the path from the root to $p[z]$. All these vertices have z in its subtree.

1.3.1 Fixing the degree of $p[z]$

We consider the following two cases.

Case A: $p[z]$ has only two leaves z_1 and z_2, and we make u a child of $p[z]$.

Subcase 1 $x < key[z_1]$

- set $t[p[z]] = s[p[z]]$ and $s[p[z]] = f[p[z]]$,
- make u the first child of $p[z]$, i.e. $f[p[z]] = u$,
- set $key[u] = x$, $f[u] = s[u] = t[u] = null$,
- set $L[p[z]] = x$ and $M[p[z]] = s[p[z]]$.

Subcase 2 $key[z_1] < x < key[z_2]$

- set $t[p[z]] = s[p[z]]$, and
- make u the second child of $p[z]$, i.e. $s[p[z]] = u$,
- set $key[u] = x$, $f[u] = s[u] = t[u] = null$,
- set $L[p[z]] = f[p[z]]$ and $M[p[z]] = x$.

Subcase 3 $key[z_2] < x$

- make u the third child of $p[z]$, i.e. $t[p[z]] = u$,
- set $key[u] = x$, $f[u] = s[u] = t[u] = null$.

Note that L or M values of some of the proper ancestors of $p[z]$ may have to be changed.

Example If we insert the element 2 into the 2-3 tree of Figure 6(a), we get the 2-3 tree of Figure 6(b).

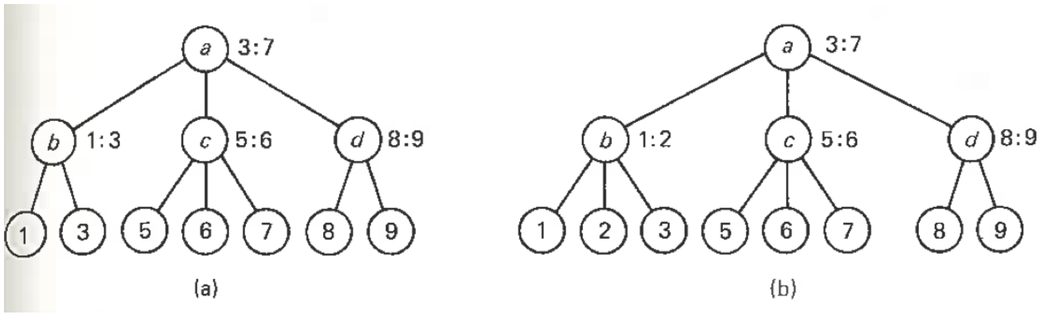


Figure 6: Insertion into a 2-3 tree: (a) tree before insertion; (b) tree after inserting 2.

Case B: $p[z]$ already has three leaves. We make u the appropriate child of $p[z]$ (i.e. the elements must be in sorted order). Vertex $p[z]$ now have four children. We need to split the vertex $p[z]$. We create a new vertex g. We associate the first two children as the children of $p[z]$, and make the two rightmost children into children of g. We then make g a sibling vertex of $p[z]$ by making g a child of $p[p[z]]$. If the parent of $p[z]$ has two children, we stop here. If the parent of $p[z]$ had three children, we must repeat this process recursively until all the vertices in the tree have at most three children. If the root node is given four sons, we create a new root with two new children, each of which has two of the four children of the former root.

Example If we insert 4 into 2-3 tee of Figure 6(a), we get Figure 7.

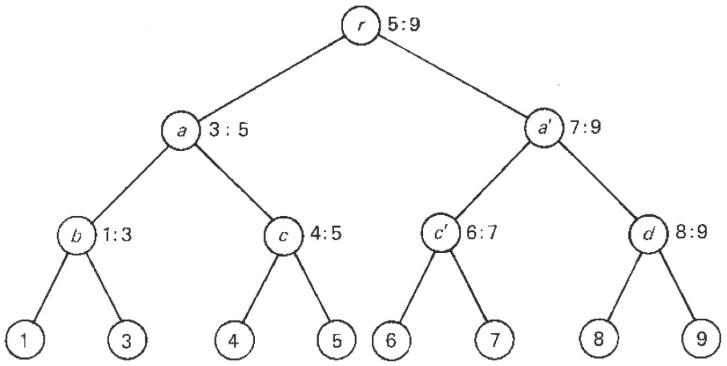


Figure 7: Tree of Fig. 6(a), after 4 is inserted.

1.4 Delete(x,r)

An element x can be deleted from a 2-3 tree in essentially the reverse of the manner by which an element is inserted. Let z be the leaf node containing x. There are three cases to consider.

1. If z is the root, remove z. In this case x was the only element in the tree.
2. If $p[z]$ (the parent node of z) has three children, remove z.
3. If $p[z]$ has two children, $f[p[z]]$ and $s[p[z]]$ (one of them is z which needs to be deleted), there are two possibilities:
    (a) $p[z]$ is the root. Remove $p[z]$ and z, and leave the remaining son as the root.

(b) $p[z]$ is not the root. Suppose $p[z]$ has a sibling vertex g to its left. A sibling to the right is handled similarly. If g has two children, make the remaining child the third child of g, remove z, and call the deletion procedure recursively to delete $p[z]$. If g has three children, make the rightmost son $t[g]$ be the left son of $p[z]$ and remove z from the tree.

Example: Let us delete element 4 from 2-3 tree 7. The leaf labeled 4 is a child of c which has two children. Thus we make the leaf labeled 5 as the rightmost child of vertex b, remove the leaf labeled 4, and then recursively remove vertex c.

Vertex c is the child of vertex a, which has two sons. The vertex a' is the right sibling of a. Thus by symmetry, we make b the leftmost child of a', remove vertex c, and the recursively remove vertex a.

Vertex a is a son of the root. Applying case 3(a), we leave a' as the root of the remaining tree, which is shown in Figure

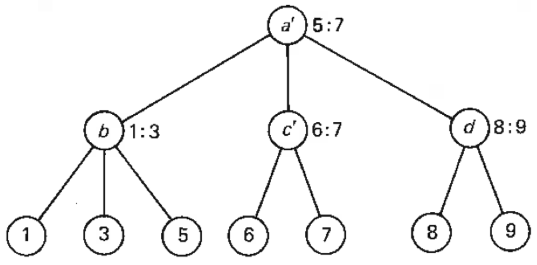


Figure 8: Tree of Fig. 7, after 4 is deleted.

We leave a formal specification of the deletion process as an exercise along with the proof that it can be executed in at most $O(\log n)$ steps on a tree with n leaves.

Remark We have seen so far that each of the instructions Insert, Delete, Min and Search can be executed on a 2-3 tree with n leaves in at most $O(\log n)$ steps per instruction when L and M values are used.

1.5 Concatenate(S_1, S_2)

The instruction **Concatenate(S_1, S_2)** takes as input two sequences S_1 and S_2 such that every element of S_1 is less than every element of S_2, and produces a merged (concatenated) sequence S_1S_2. We assume that S_i is represented by 2-3 tree $T_i, i = 1,2$. The sorted sequences are stored at the leaves.

The merging process algorithm can be described as follows.

Case 1: Height(T_1) = Height(T_2):
- Create a new root r;
- Make $Root[T_1]$ and $Root[T_2]$ the first and the second son of r;
- Determine L and M values of r.

Case 2: Height(T_1) > Height(T_2):
- Let v be the vertex on the rightmost path of T_1 such that $Depth(v) = Height(T_1) - Height(T_2)$;
- $p[v]$ is the parent node of v. Make $Root(T_2)$ a child of $p[v]$ immediately to the right of v;
- If $p[v]$ has four children, fix the structure using the method similar to the insertion operation.

Case 3: Height(T_1) < Height(T_2): The treatment is very similar.

Lemma 1.4 *$Concatenate(S_1, S_2)$ can be implemented in $O(logn)$ time. We assume that S_1 and S_2 are available in 2-3 trees. The result is left in a 2-3 tree, and every element of S_1 is smaller than every element of S_2.*

2 Hashing

A hash table is an effective data structure for implementing dictionaries. Hashing has $\Theta(n)$ worst case complexity, but performs extremely well in practice. We will see that number theory plays an important role in the design of hash functions.

Hashing describes a way to store the elements of a set in a table by breaking the universe of possible keys. The position of the data element in the table is given by a function from the key. This can be visualized in Figure 9.

A sample of an implementation of hash table in JAVA is given in Figure 10.

The following issues we should be aware of:

Size of the hash table: Only a small subset of all possible keys of the universe we need to store.

Address calculation: The keys are not necessarily integers. The index of the table being computed depends on the the size of hash table.

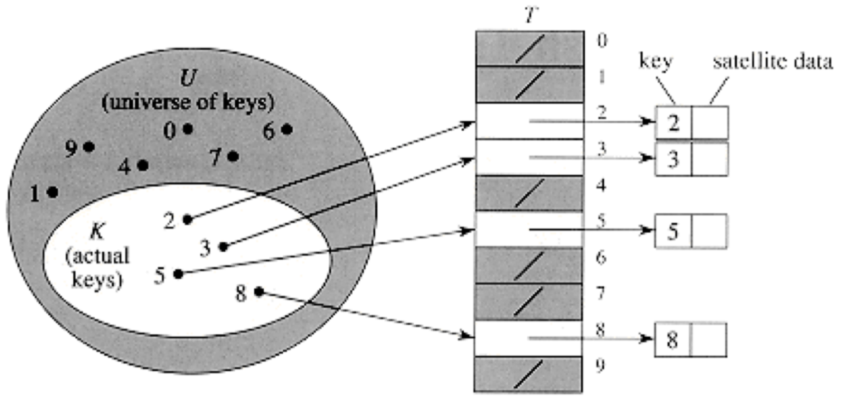


Figure 9: The universe of **key** is divided into m containers (buckets). The figure is tan from CLRS.

The hashing process can be visualized as follows Figure 11.

Definition Let U be the universe of all possible keys, and $\{B_0, B_1, B_2, \ldots, B_{m-1}\}$, a set of m buckets for storing the elements from U. The **hash function** is a mapping

$$h : U \rightarrow \{0, 1, 2, \ldots, m-1\},$$

mapping each key $s \in U$ to a number $h(s)$.

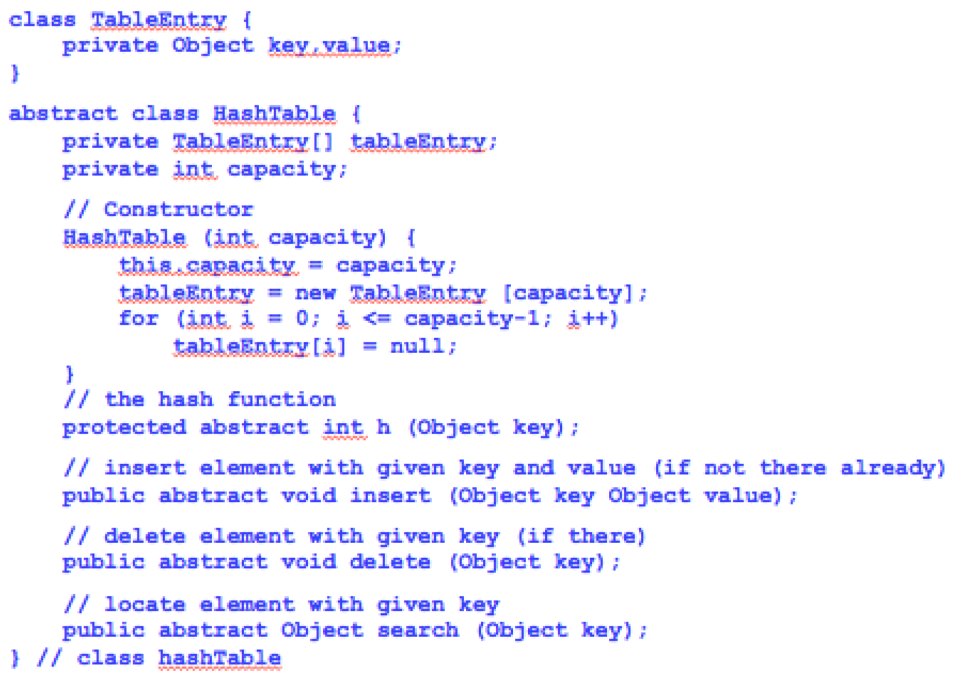
The respective key goes to bucket $B_{h(s)}$. The bucket number $h(s)$ is also called **hash address**. The complete set of buckets is called **hash table**.

2.1 Examples

1. The **division method** $h(k) = k \bmod m$ is a simple hash function that can be used when the keys of the records are integers. The table size m must be chosen carefully.

   How to choose m?

   - if m is even and $h(k)$ is even, keys will be stored in even locations. It is not a good idea.
   - $m = 2^p$ yields the bucket number using the lower p bits of k.



```
class TableEntry {
    private Object key,value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    // Constructor
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry [capacity];
        for (int i = 0; i <= capacity-1; i++)
            tableEntry[i] = null;
    }
    // the hash function
    protected abstract int h (Object key);

    // insert element with given key and value (if not there already)
    public abstract void insert (Object key Object value);

    // delete element with given key (if there)
    public abstract void delete (Object key);

    // locate element with given key
    public abstract Object search (Object key);
} // class hashTable
```

Figure 10: Implementation of hash table in JAVA. The slide is taken from the lecture notes of Prof. Th. Ottmann.

- Selecting m to be a prime number not close to a power of 2 is typically a good choice.

Rule: Choose m prime not too close to the exact power of 2 is often a good choice.

Suppose we wish to allocate a hash table to hold roughly $n = 2000$ character strings, where a character has 8bits. Suppose we don't mind examining an average of 3 elements in an unsuccessful search. We allocate a hash table of size 701. This number is chosen becausee it is a prime near $2001/3$, but not a power of 2. Therefore, our hash function is $h(k) = k \bmod 701$.

2. The **multiplicative method** is another simple rule. First the key (in base 2) is multiplied by a constant value A such that $0 < A < 1$, and $p = \log_2 m$ bits are selected for the hash function value from somewhere in the fractional part of the resulting base-2 numeral (generally the leftmost p bits). Random numbers are generated in a similar manner.

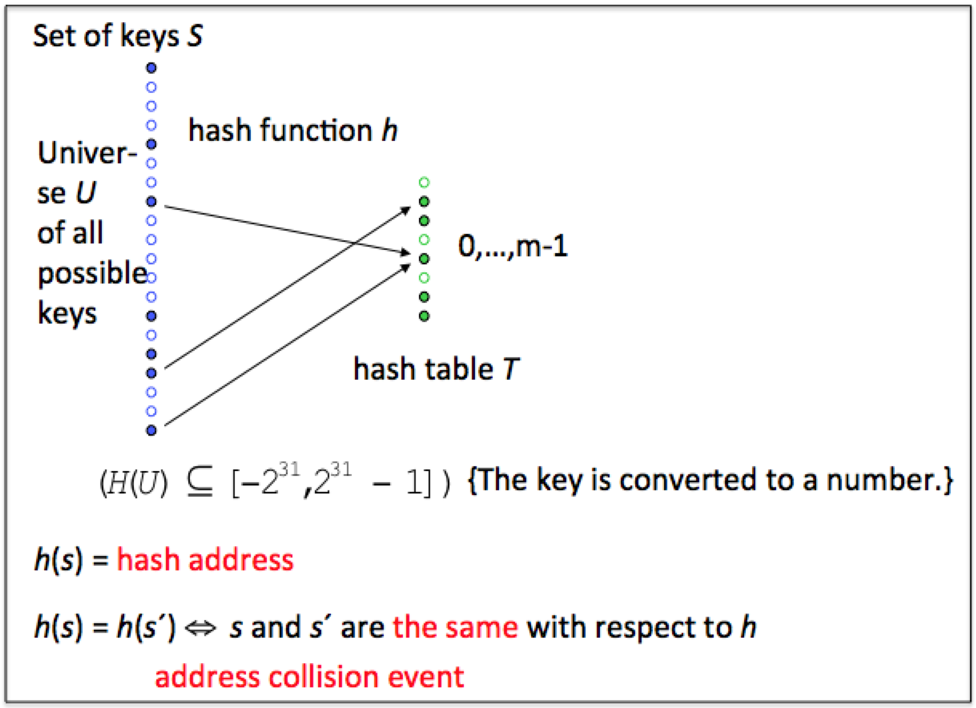

Set of keys S
hash function h
Univer-se U of all possible keys
0,...,m-1
hash table T
$(H(U) \subseteq [-2^{31}, 2^{31}-1])$ {The key is converted to a number.}
$h(s)$ = hash address
$h(s) = h(s') \Leftrightarrow$ s and s' are the same with respect to h
address collision event

Figure 11: Hashing process

Consider a hash table of size $m = 16$. The address size $p = \log m = 4$ bits and keysize $w = 6$ bits. Let $A = .101011_2$ be the fractional constant. Given a key $k = 011011_2$, we calculate $A.k = 010010.001001_2$. Then take $i = .0010_2$, low end four bits in the fractional part, as the index of the hash table. It is suggested by Donald Knuth (The Art of Computer Programming, Vol. 3) A=0.6180339887... as a good choice as a multiplier. He also suggests that the value of m, the table size should be a prime number

2.2 Possible ways of treating collisions

Collisions are treated differently in different methods. A data object with key s is in collision with another object with key s' id $h(s) = h(s')$, i.e. both objects have been mapped to the same index of the hash table.

There are two common methods for resolving collisions:

Chaining: Here each bucket is maintained as a linked list. All the objects whose keys are mapped to the same bucket are chained together. The search for a key is nothing but searching a specific linked list.

Open addressing: Colliding elements are stored in other vacant buckets in the table. A collision resolution scheme assigns a sequence $\alpha_0 = h(s), \alpha_1, \alpha_2, \ldots$ of addresses to each key. The insertion algorithm inspects the cell in the order indicated until it finds an empty slot in the table.

2.2.1 Analysis

We have a hash table T with m slots that stores n objects. In the worst case, the bound for a key search is $O(n)$. However, hashing method has an extremely good average case behaviour under **simple uniform hashing**. Under this assumption, any given key of the universe is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

We define the **load factor** α for T as n/m. Let n_j denote the length of the chain $T[j]$ so that $n = \sum_{j=0}^{n} n_j$. The expected value of n_j is $E[n_j] = \alpha = n/m$.

During the search for a key, the amount of work involved is

- Compute the hash value which costs $O(1)$, and
- Search a list of expected size α

Theorem 2.1 *For the chaining resolution scheme, the expected cost of an unsuccessful search is $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.*

For a successful search, the analysis is slightly different, but the bound is still the same (see CLRS: Theorem 11.2). Therefore,

Theorem 2.2 *Under the assumption of uniform hashing, the expected cost of each dictionary operation is $\Theta(1+\alpha)$ for the chaining resolution scheme. This cost is constant if $n \in O(m)$.*

In open-address hashing, n must be less than m. Therefore, the load factor $\alpha < 1$. Please refer to section 11.4 of CLRS for the details of the following two theorems.

Theorem 2.3 *Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected cost of an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.*

Theorem 2.4 *Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most*

$$\frac{1}{\alpha} \log_e \frac{1}{1-\alpha},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched.