Prepared by: [Arun Konagurthu]

Source material acknowledgement

These lecture slides are built on the [source material] developed by [Lloyd Allison].

FIT2004 S1/2017: Algorithms and Data Structures

Prerequisite material: **(1) Hashing – Linear and Quadratic probing (2) Binary searh trees**

Faculty of Information Technology, Monash University

What is covered in these slides?

Prerequisites covered already in FIT1008

- Collision resolution using Linear chaining
- Collision resolution using Linear and Quadratic probing
- Binary search trees and basic operations on them.

Recommended reading

- Hashing:
  http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Table/
- Weiss "Data Structures and Algorithm Analysis" (Chapter 4&5's relevant sections)
- Search Trees part of
  http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/

Closed-addressing using **Linear chaining**

One form of collision resolution requires putting keys that collide in a linear **list** associated with index.

```
/*LINEAR CHAINING EXAMPLE ON SOME FIRSTNAMES
  N (number of keys) = 27, M (table size) = 11
  hash function used:
  hash("ARUN") =
  (((ascii('A')*3 + ascii('R'))*3 + ascii('U'))*3 + ascii('N'))*3 % M
*/
       0: WILLIAM --> JASON
       1: BENJAMIN --> SOPHIA --> BENJAMIN
       2: DANIEL --> CAMPBELL --> DANIEL --> LACHLAN
       3: MAXIMILIAN --> SITGGEN --> JESSE
       4: SAMUEL --> LUIZA --> PETER
       5: EMMANUEL --> CHRISTOPHER --> ADITI
       6: JOSHUA
       7: ALI --> ALEXANDER
       8:
       9: MATTHEW --> DEMETRIOS --> MATTHEW
      10: LAWSON --> JEREMY --> MARTIN
```

Closed-addressing using **Linear chaining**

One form of collision resolution requires putting keys that collide in a linear **list** associated with index.

```
/*LINEAR CHAINING EXAMPLE ON SOME FIRSTNAMES
  N (number of keys) = 27, M (table size) = 11
  hash function used:
  hash("ARUN") =
  (((ascii('A')*3 + ascii('R'))*3 + ascii('U'))*3 + ascii('N'))*3 % M
*/
       0: WILLIAM --> JASON
       1: BENJAMIN --> SOPHIA --> BENJAMIN
       2: DANIEL --> CAMPBELL --> DANIEL --> LACHLAN
       3: MAXIMILIAN --> SITGGEN --> JESSE
       4: SAMUEL --> LUIZA --> PETER
       5: EMMANUEL --> CHRISTOPHER --> ADITI
       6: JOSHUA
       7: ALI --> ALEXANDER
       8:
       9: MATTHEW --> DEMETRIOS --> MATTHEW
      10: LAWSON --> JEREMY --> MARTIN
```

What is the worst-case time complexity for standard operations on hash-tables: **insert("key"), delete("key"), lookup("key")?**

Open-addressing using **Linear probing**

- Linear probing is the simplest probing strategy.
- Generalized formula for ith probe after collision

$$\textbf{hash}(\text{“}\textbf{key}\text{”}, i) = (\textbf{hash}(\textbf{“key”}) + c_1 i) \pmod{M}$$

where, c_1 is the increment or step-size of probe
M is the size of the hash table.

Linear probing example

M = 8

Empty hash table

0	1	2	3	4	5	6	7
????	????	????	????	????	????	????	????

Linear probing example

M = 8

Empty hash table

0	1	2	3	4	5	6	7
????	????	????	????	????	????	????	????

insert("fred"); Let **hash("fred") = 6**

0	1	2	3	4	5	6	7
????	????	????	????	????	????	fred	????

Linear probing example

M = 8

Empty hash table

0	1	2	3	4	5	6	7
????	????	????	????	????	????	????	????

insert("fred"); Let **hash("fred") = 6**

0	1	2	3	4	5	6	7
????	????	????	????	????	????	fred	????

insert("anne"); Let **hash("anne") = 2**

0	1	2	3	4	5	6	7
????	????	anne	????	????	????	fred	????

Linear probing example

M = 8

Empty hash table

0	1	2	3	4	5	6	7
????	????	????	????	????	????	????	????

insert("fred"); Let **hash("fred") = 6**

0	1	2	3	4	5	6	7
????	????	????	????	????	????	fred	????

insert("anne"); Let **hash("anne") = 2**

0	1	2	3	4	5	6	7
????	????	anne	????	????	????	fred	????

insert("jim"); Let **hash("jim") = 2** -- collision!

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

Linear probing example – cont'd

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

Linear probing example – cont'd

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

probe for empty **hash+1, hash+2, ...** **(mod M)**

0	1	2	3	4	5	6	7
????	????	anne	jim	????	????	fred	????

Linear probing example – cont'd

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

probe for empty **hash+1, hash+2, ...** **(mod M)**

0	1	2	3	4	5	6	7
????	????	anne	jim	????	????	fred	????

insert("jill"); Let **hash("jill") = 2** -- collision!

0	1	2	3	4	5	6	7
????	????	anne jill	jim	????	????	fred	????

Linear probing example – cont'd

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

probe for empty **hash+1, hash+2, ...** **(mod M)**

0	1	2	3	4	5	6	7
????	????	anne	jim	????	????	fred	????

insert("jill"); Let **hash("jill") = 2** -- collision!

0	1	2	3	4	5	6	7
????	????	anne jill	jim	????	????	fred	????

probe... collision!

0	1	2	3	4	5	6	7
????	????	anne	jim jill	????	????	fred	????

Linear probing example – cont'd

0	1	2	3	4	5	6	7
????	????	anne jim	????	????	????	fred	????

probe for empty **hash+1, hash+2, ...** **(mod M)**

0	1	2	3	4	5	6	7
????	????	anne	jim	????	????	fred	????

insert("jill"); Let **hash("jill") = 2** -- collision!

0	1	2	3	4	5	6	7
????	????	anne jill	jim	????	????	fred	????

probe... collision!

0	1	2	3	4	5	6	7
????	????	anne	jim jill	????	????	fred	????

probe further...

0	1	2	3	4	5	6	7
????	????	anne	jim	jill	????	fred	????

Problem with Linear chaining

The problem with linear probing is that collisions from **nearby hash values** tend to merge into **big blocks**, and threfore the lookup can degenerate into a linear $O(N)$ search.

Quadratic probing is a more sophisticated probing strategy!

$$\textbf{hash}(\text{“}\textbf{key}\text{''}, i) = (\textbf{hash}(\textbf{“key''}) + c_1 i + c_2 i^2) \pmod{M}$$

When $c_1 = c_2 = \frac{1}{2}$, the probe sequence after collision is **`+1, +3, +6, ...`**
Note, when $c_2 = 0$, this becomes linear probing!

Quadratic probing is not guaranteed to probe every location in the table – an insert could fail while there is still an empty location. However hash tables are rarely allowed to get full – we will see this in a later slide. **The same probing strategy is used in the associated search routine!**

Advantage of Quadratic probing over linear

- The virtue of quadratic probing is that the probe locations for two near-miss keys, as opposed to two colliding keys, are not closely related.
- this reduces the likelihood of large collision blocks forming, which is a big problem with linear probing that slows down the search process.
- For example, if one set of keys all hash to '6' and another set hash to '7' then the collision blocks of these sets will over-run and merge with each other under linear-probing.
- With quadratic probing on the other hand, the probes after 6 are at positions 7, 9, 12, 16, ... and the probes after 7 are at positions 8, 10, 13, 17, ... This tends to give small disjoint blocks.

Testing the performance of linear and quadratic probing

Simulations give an estimate of the performance of a hash table based on its **Load** (ratio of number of hashed keys to table size). The time taken by **insert** and **lookup** depends on **collisions/lookup**.

Linear probing		Quadratic probing	
Load	**collisions/lookup**	**Load**	**collisions/lookup**
10%	0.04	10%	0.04
20%	0.09	20%	0.10
30%	0.15	30%	0.15
40%	0.27	40%	0.25
50%	0.39	50%	0.38
60%	0.68	60%	0.53
70%	0.92	70%	0.67
80%	1.72	80%	1.09
90%	2.29	90%	1.40

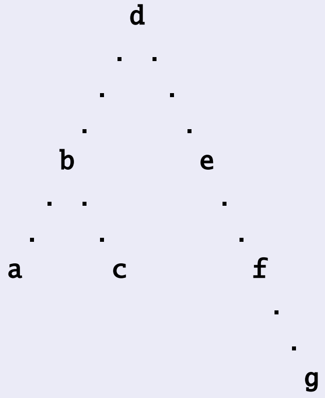
Results here use hash keys that are uniform random numbers. This is the best possible case; Performance in real setting is actually worse.

Binary Search Trees (BST): Introduction

- The empty tree is a BST
- If the tree is not empty,
  1. the elements in the **`left subtree`** are **LESS THAN** the element in the root
  2. the elements in the **`right subtree`** are **GREATER THAN** the element in the root
  3. the **`left subtree is a BST`**
  4. the **`right subtree is a BST`**

Note! **Don't forget last two conditions!**

BST example



Binary Search Trees (BST): Introduction

BST example

```
/*Input:*/ the land rover is a type of motor car

                         the
                         . .
                        .   .
                    land    type
                     . .
                    .   .
                  is       rover
                  .         .
                 .         .
                a       of
                 .      .
                  .    .
                 car  motor
```

Searching for some **x** in the BST

```
// BST implemented here as a tree data structure
//     T = fork(e, L, R)

function search(x,T)
   if (T == nilTree)
     return false     // not present!

   else if ( x < e ) // search x in Left subtree
     search( x , L )

   else if ( x > e ) // search x in Right subtree
     search( x , R )

   else  return true        // found!
```

Inserting **x** into a BST

```
// BST implemented here as a tree data structure
//     T = fork(e, L, R)
function  insert( x , T )
   if (T == nilTree)  // Insert here as leaf node
     T = fork( x , nilTree , nilTree )

   else if ( x < e )  // Traverse and insert ...
     insert( x, L );  //  along the Left subtree


   else if (x > e )   // Traverse and insert ...
     insert( x , R )  //  along the Right subtree

   else      // x == e
     ... it depends ... // [discussed in lecture]
return
```

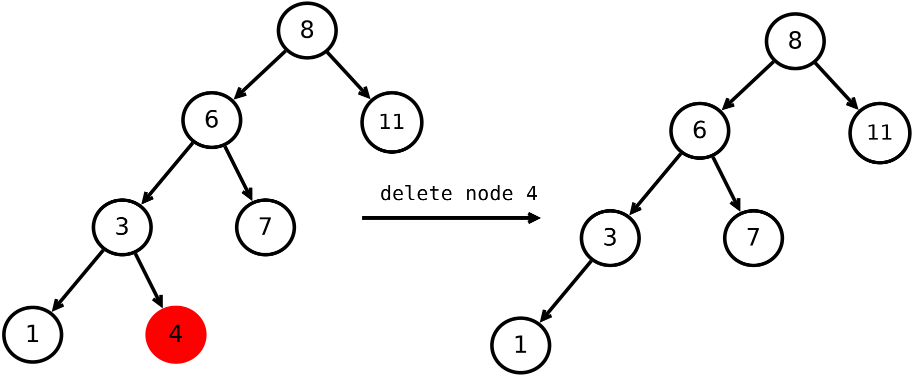
Delete **x** from a BST

```
First lookup x in T // Check if x is present in T

Assuming x found, there are three cases:

    case 1
        No children, x is a leaf node.
            Easy case! freeup leaf;
                       set subtree to nilTree;

    case 2
        x has One child
           Fairly easy case!

    case 3
        x has Two children.
           Tricky case!
```

Delete **x** from a BST – Case 1 (example)

Case 1

```
No children, x is a leaf node.
    Easy case! freeup leaf; set subtree to nilTree;
```

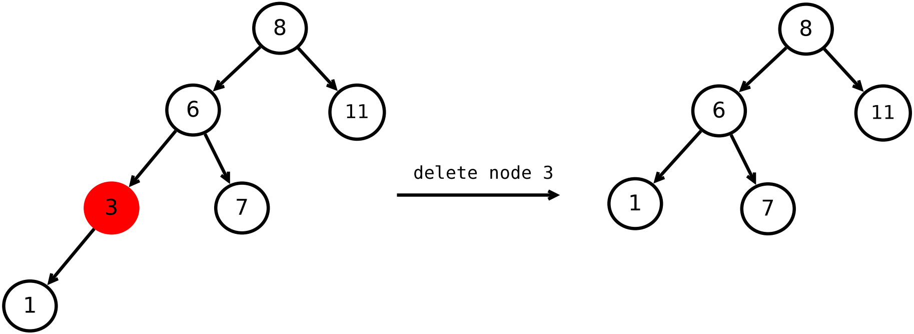


Delete **x** from a BST – Case 2 (example)

Case 2

One child

fairly easy . . .



Delete **x** from a BST – Case 3

Case 3

```
 Two Children        .
                      .
                       x
                      . .
                     .   .
                    T1   T2

Deleting node x requires some thought ...

Find smallest element, y, in right tree, T2
[OR  greatest element, y, in left  tree, T1]

overwrite x with y,

delete    y from T2!    [or y from T1!]
```

Delete **x** from a BST – Case 3 (example)

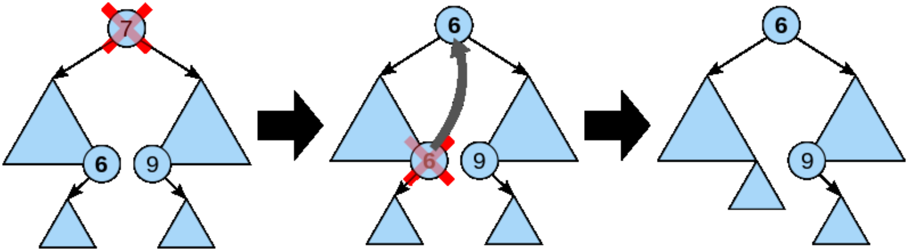


Image source: wikipedia

Delete **x** from a BST – pseudocode

```
 // BST implemented here as a tree data structure
 //    T = fork(e, L, R)
Tree delete(x, T)
   if(T != nilTree)
       if( x < e )        L = delete(x, L);
       else if( x > e )   R = delete(x, R);
       else /* found x: e == x */
       if(L  == nilTree)      /* left tree is empty  */
           T=R;
       else if(R == nilTree)  /* right tree is empty */
          T=L;
       else                   /* neither subtree is empty */
          findAndDel(T);  /* described in next slide */
     }
return T; /* also see over page . . . */
```

Delete **x** from a BST – pseudocode

```
// BST implemented here as a tree data structure
//    T = fork(e, L, R); T.elt   = root node/element
//                       T.right = right subtree
//                       T.left  = left subtree
function findAndDel(T)
/* find smallest element in right subtree, copy it
  to the root node and delete the original. */
  Tree R = T.right;
/* one step right, then */

  while( R.left != null )
/* as far left as possible */
     R = R.left;

  T.elt = R.elt;                          /* copy */
  T.right = delete(T.elt, T.right); /* delete original! *
  return;
}
```

BST summary

- BST is a kind of Lookup table – previously we studied hash table
- A lookup table is an Abstract Data Type that supports operations like **`search/lookup`**, **`insert`**, **`delete`**, ...
- Complexity
  - **`lookup`**, **`insert`**, **`delete`** **proportional to height of the tree**
  - $O(n)$-time, **worst-case**
- Note: **Minimum element**, **`y`**, is the **left-most** in BST
- Note: **Maximum element**, **`y`**, is the **right-most** in BST