Dynamic Data Structures

- fixed size, but much smaller than the full data set
- expandable in increments
- array doubling
- size of the data structure?
- data that isn't known about in advance
- online data rather than static data
- symbol tables
- large data sets to be searched

Why do we need dynamic data structures?
Note: Most of these operations pass a key value in and expect a pointer returned.

- Predecessor (of an element in a set)
- Successor (of an element in a set)
- Maximum (element in a set)
- Minimum (element in a set)
- Delete (an element from a set)
- Insert (an element into a set)
- Search (for a key value in a set)
\[
\sqrt{\frac{2}{\sqrt{2}}} \approx 1.414
\]

9.3.2 Good Hash Functions

9.3.1 Basic Issues in Hashing

Distinct from, but not unrelated to, hashing for cryptography:

- Good for present/absent tests
- Good for retrieving large sets of keys without sorting
- If the keys are clustered, hashing can randomize the effective key
- If the key is large relative to the number of data items, hashing can be useful

Division function:
\[ y \leftarrow y \pmod{m} \]

Multiplication function:
\[ k[A \pmod{1}, \ldots, k[A \pmod{m}] \]

Hash function

Draining or open addressing? (page 225)

Collisions
9.4 References

\[
\left( \left( y \mod m \right) \cdot y \right) + 1 = (y^2) \mod m
\]

and

\[
\left( \left( y \mod m \right) \cdot y \right) = (y^2) \mod m
\]

Less than \( m \):

- Double Hashing:
  
  \[
  \left( \left( y \mod m \right) \cdot y \right) \mod m, \text{ for the } i\text{-th probe}.
  \]

- Quadratic Probing:
  
  \[
  \left( \left( y \mod m \right) \cdot y \right) \mod m, \text{ for the } i\text{-th probe}.
  \]

- Linear Probing:
  
  \[
  \left( y \mod m \right) \text{ for the } i\text{-th probe}.
  \]

How to deal with collisions? 

What's the probe sequence?

9.3.3 Open Address Hashing
If would be nice to be able to create balanced trees in an online situation, but we can't always do that.

Definition 9.1 A binary search tree is called a binary search tree if the key at each node is greater than all the keys in its left subtree and less than or equal to all the keys in its right subtree. In this case the tree is called a balanced binary search tree.
traversal

Note that if we have a binary search tree, then we can print the elements in order by doing an inorder traversal.

• print current key
• recursive right
• recursive left

Postorder traversal

• recursive right
• recursive left
• print current key

Preorder traversal

• recursive right
• recursive left
• print current key

Inorder traversal
Balanced binary trees are clearly great, but if we had to do dynamic insertion, we could wind up with this.
{ 
  { 
    
    return(search-right[x], k) 
  } 
  else 
  { 
    return(search-left[x], k) 
  } 
} 

if (k > key[x]) 
else 
{ 
  return(x)! 
} 

if (key[x] == NULL) 
  return(true search(x), k) 

All the basic ops are O(h) on a binary search tree of height h.

9.5.1. Operations on Binary Search Trees
{ 
    return 
    
    { 
        { 
            [x][x] = return 
        } 
    } 
    else 
    { 
        [x][x] = return 
    } 
    if (key > key) 
    { 
        (([x][x] = key) || (key = key)) 
    }
}

void tree-search(x, k) 
   SEARCH // or else
{ }

return (x)?
{

right[x]\[right]\[x\[right] = x
}

while(right[x]\[right] = NIL)

(x) //

(void tree-maximum)

{ }

return (x)?
{

left[x]\[left]\[x\[left] = x
}

while(left[x]\[left] = NIL)

(x) //

(void tree-minimum)
{ }
{ }
return (λ) { }
{ }
[λ] parent = λ
[λ] x = x

{ ([λ] right = NIL (x) \& x \neq \lambda) \rightarrow
[λ] parent = x

else
{ }
return (tree-minimum(right[x]) )

( ([right[x] = NIL (x) \& x \neq \lambda) \rightarrow

void (tree-successor(x))

//


```c
)
else
{
    z = [T]root

}

if NIL ==  ∧
∧ = [Z]parent
{
    {

    if x left = [x]

    }

    else
    {

    if x right = [x]

    }

    if [x] key > [Z] key

    x = ∧

    }

while (NIL = x)
{
    [L] root = x

    NIL = ∧

}

void tree-insert(T, z)
NIL = [Z] right, ∨ = [Z] key, left = NIL
INSETION node z, key = [Z] key, node = NIL
```
```c
{
    { 
        z = [\^]v,
        if (key > [\^]z[\^]key)
        { 
            z = [\^]v,
        }
    }
}
```
will use some of each."

Algorithms, Addison-Wesley, chapter 13. The first edition was published in 1983 and the second in 1988. I will do the former, simply out of personal preference. I will follow closely the development in Sedgewick, which is the same as the red-black tree. The text describes the latter and then does the former in an exercise.

To deal with this, there have been dozens (hundreds?) of methods developed for balancing trees on the fly. Loaded has already been sorted by SSN, and thus that the search tree will be horribly skewed.

is inserting Social Security numbers, for example, into a tree, then it's entirely possible that the data to be inserted is already been sorted by SSN, and thus that the search tree will be horribly skewed.

But, of course, it's hard to guarantee that the load, insert, and delete operations will be providing "random" key data. If one

searched based on some key. With a complete balanced tree one gets all items stored for a cost of \( O(n) \) for

Clearly, one can't really do better than a complete balanced binary tree for storing data that has to be searched
Trees.

already a 4-node.

were C, then we must do something different in order to insert the new key into the part of the tree that is done. Similarly, a leaf 3-node can be turned into a leaf 4-node by inserting the new element. But if the key node, and if terminatizes in a leaf node that is a 2-node, we just turn the leaf 2-node into a leaf 3-node and are inserted is only slightly more complicated. If we do the search algorithm to find where to insert the new

An algorithm to search such a tree is an easy extension of the standard binary search tree algorithm.

\[ * * * * * * * * \]

\[ S \quad H \quad N \quad I \]

\[ V \quad A \quad C \]

\[ E \quad R \]

the NIL pointer.

So, for example, a more generic 2-3-4 tree might look like the following, where we interpret the box to be
could have put the I in either the node or the N node.

Note that the leaf-breaking rule would have been invoked if we were inserting another I and not G, since we

* * * * * * * * *

So inserting G would result in the following tree:

- Break the leaf randomly or by defaulting to either the left or right 2-node.
- Insert the new element as needed into one of the two 2-nodes.
- Split the 4-node into two 2-nodes and pass the middle element to the parent.

What we do is the following:
We call these top-down &-trees.

the way back to the root, we propagate a split up one level at a time the next time we encounter a 4-node.

Another way to view this is that instead of waiting until we have no choice but to propagate the split all
4-nodes.

It means that the only 4-nodes that will exist in the tree are either leaf nodes (those do not have children)
The solution to this is that any 4-node that happens to be encountered in the search down the tree is split.

demands parent to parent all the way up to the root is Le n time.

then propagate through the entire tree. (Propagating an element up to the parent is consistent time; pushing
of splitting all the way back to the root and thus be inefficient. We would like all actions to stay local rather
already be a 4-node and have to be split. This wouldn’t be a good thing, since it would force the possibility
created another problem, which is that when we pass the middle element up to the parent, the parent may
Now, we have to deal with the fact that in solving one problem, how to add elements to 4-nodes, we have
Theorem 10.2. Insertions in top-down $2\times 4$-trees of $n$ nodes require more than $\log n + 1$ splits.

Theorem 10.1. Searches in top-down $2\times 4$-trees of $n$ nodes never visit more than $\log n + 1$ nodes.

Note that if we are splitting as we search down the tree, then these local splits will be all that are needed:

\[
\begin{align*}
&\text{E G I J K} \\
&\quad \text{A C F H J} \\
&\quad \quad \text{B D H} \\
&\quad \text{A C F H} \\
&\quad \quad \text{B D} \\
&\text{C E G I} \\
&\quad \text{A C D H} \\
&\quad \quad \text{B F} \\
&\quad \text{A D H} \\
&\quad \quad \text{B F} \\
&\text{E G I} \\
&\quad \text{A C F H} \\
&\quad \quad \text{B D} \\
&\text{E G I} \\
&\quad \text{A C F H} \\
&\quad \quad \text{B D} \\
\end{align*}
\]
we would split the 4-node and convert the tree to

For example, the next time we started a search in the tree
We have two choices as to which way to point the 3-nodes. What we do is color the arcs red and black and label the nodes with the color of the link pointing to that node. We need to implement the data structures necessary to implement 2-nodes, 3-nodes, and 4-nodes would be a pain. So we implement conceptual, then, these are balanced trees we want to implement. However, during the manipulations on
Any child of a red node is black.
The number of black arcs from the root to any leaf is constant.
Every leaf is black.
The root is black.
Every node is either red or black.