CS F211
Data Structures & Algorithms

Dictionary Data Structures – Search Trees

Comparison of Sorted Arrays and Hashtables

Ordered Dictionaries

- Better Representation
- Binary Trees
- Binary Search Trees
  - Implementation (Find, Add, and Delete)

Dictionary Implementations - Comparison

Sorted Array

- Suitable for:
  - Ordered Dictionary
    - Example Queries: 2^{nd} largest element? OR the element closest to k?
  - Offline operations (insertions/deletions)
  - Comparable Keys
- Implementation:
  - Deterministic

Hashtable

- Suitable for:
  - Unordered Dictionary
  - Online insertions (deletions??)
    - Resizing can be done at an amortized cost of O(1) per element
  - Hashable Keys
- Implementation:
  - Randomized

Dictionary implementations - Comparison

Sorted Array

- Time Complexity (find):
  - ϴ(logN) - worst case and average case
- Space Complexity
  - ϴ(1)

Hashtable

- Time Complexity (find):
  - ϴ(1) average case and ϴ(N) worst case
- Space Complexity
  - ϴ(N) words – separate chaining (links)
  - ϴ(N) bits – open addressing (empty & deleted flags)
  - When rehashed?

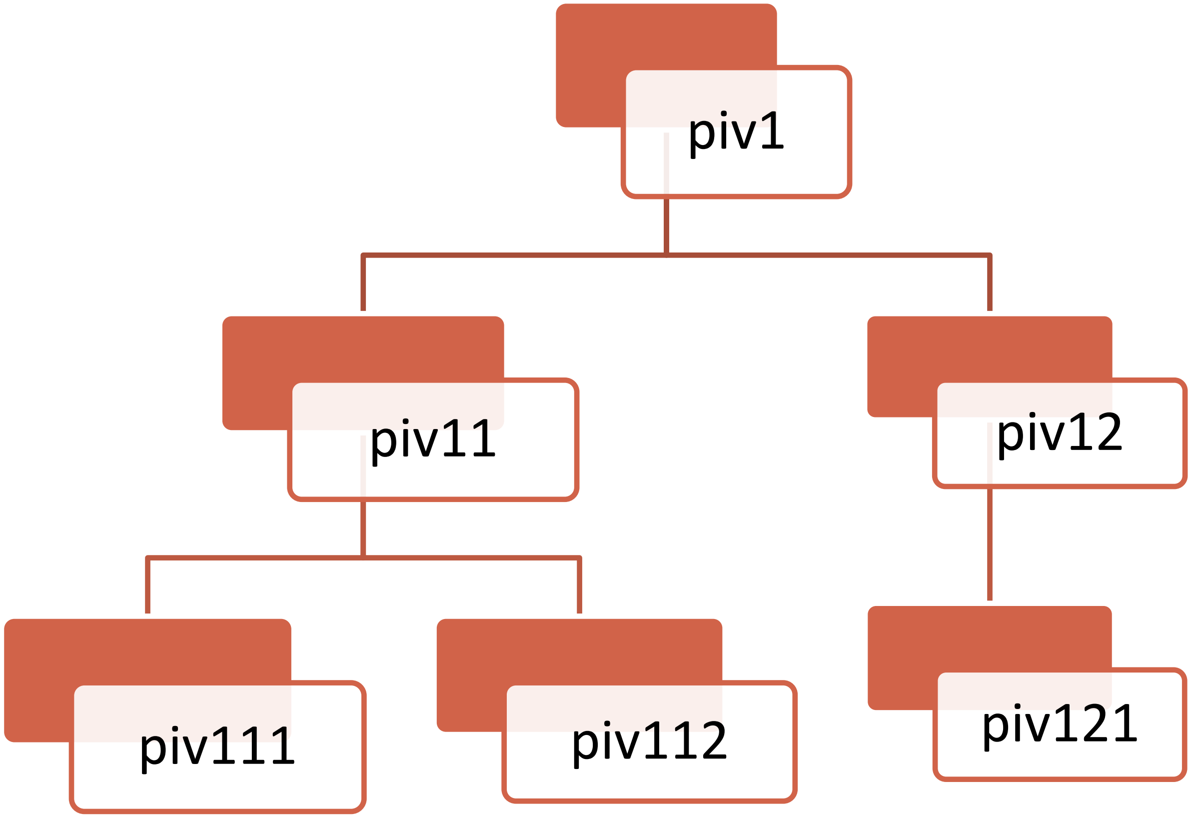
i.e. not fully incremental!

ORDERED DICTIONARY – BETTER REPRESENTATION?

- Is there an representation that
  - supports "relative order" queries and
  - supports online operations and
  - is (incrementally) resizable ?
- Revisit (the general structure of) Quicksort(Ls)

```
Quicksort(Ls) {
 If (|Ls|>0) {
  partition Ls based on a pivot into LL and LG
  QuickSort LL
  QuickSort LG
 }
}
```

QuickSort Visualized



Ordered Dictionary – Better Representation?

- Can we re-materialize the *QuickSort order* while searching?
  - i.e. a representation where ***key*** is compared with the ***pivot***
    - key == pivot ==> done
    - key < pivot ==> search in left subset
    - key > pivot ==> search in right subset.
- This is similar to QuickSelect but
  - With stored "ordering" between the pivots.
    - i.e. ordering is preserved after sorting so as to support to "relative order" queries

ORDERED DICTIONARY – BETTER REPRESENTATION?

- Data Model:
  - A Set is characterized by the “Relation between Pivot and two (sub)sets”
- Generalized Data Model:
  - A set is characterized by a “root” element and two subsets.

ORDERED DICTIONARY – BETTER REPRESENTATION?

- Inductive Definition:
  - A ***binary tree*** is
    1. empty OR
    2. made of a root element and two **binary trees** - referred to as **left** and **right (sub) trees**
- For induction to be well founded “sub trees” must be of smaller size than the original.
- Sub trees are referred to as ***children*** (of the node which is referred to as the ***parent***)
  - A binary tree with two empty children is referred to as a ***leaf***.

- Inductive Definitions can be captured recursively:

BinaryTree = EmptyTree U (Element x BinaryTree x BinaryTree)

ADT Binary Tree

- BinaryTree createBinTree() // create empty tree
- Element getRoot(BinaryTree bt)
- BinaryTree getLeft(BinaryTree bt)
- BinaryTree getRight(BinaryTree bt)
- BinaryTree compose(Element root,
  BinaryTree leftBt,
  BinaryTree rightBt)

ADT Binary Tree - Representation

- struct __binTree;
- typedef struct __binTree *BinaryTree;
- struct __binTree { Element rootVal;

```
struct __binTree {   Element rootVal;
                     BinaryTree left;
                     BinaryTree right;
                 };
```

Argue that the above representation in C captures the definition:

BinaryTree = EmptyTree U (Element x BinaryTree x BinaryTree)

ADT Binary Tree - Implementation

```
BinaryTree compose(Element e, BinaryTree lt, BinaryTree rt)
{
   BinaryTree newT =
          (BinaryTree)malloc(sizeof(struct   __binTree));
    newT->rootVal = e;
    newT->left = lt;
    newT->right = rt;
    return newT;
 }
```

Ordered Dictionary – Search Tree

- A binary search tree is
- a binary tree that captures an "ordering" (i.e. a relation) $\lesssim$ via the relation between the root and its subtrees:
  - i.e. for each element *eL* in the left subtree:
    - *eL* $\lesssim$ *rootVal*
  - and for each element *eR* in the right subtree:
    - *rootVal* $\lesssim$ *eR*

ADT Ordered Dictionary

- Element find(OrdDict d, Key k)
- OrdDict insert(OrdDict d, Element e)
- OrdDict delete(OrdDict d, Key k)
  - Note on Representation:
    - We can use the same BinaryTree representation for this.
      - i.e. The ordering is captured implicitly at the point of insertion by leveraging the left and right information.
    - Hence the following type definition – in C – would serve as the data definition!
  - End of Note.
- typedef BinaryTree OrdDict;

ADT Ordered Dictionary – Implementation

```
//Preconditions:   k is unique;
Element  find(OrdDict d, Key k)
{
   if (d==NULL) return NOT_FOUND;
   if (d->rootVal.key == k) return d->rootVal;
    else if (d->rootVal.key < k) return find(d->right, k);
    else /* d->rootVal.key > k */  return find(d->left, k);
}
```

Exercise: Modify implementation for multiple elements with the same key value.

BST: Implementation of Insert

```
//Preconditions:  d is non-empty;  keys are unique (i.e. duplicates);
OrdDict  insertNE(OrdDict d, Element e)
{
  if (d->rootVal.key < e.key)  {
    if (d->right == NULL) { d->right = makeSingleNode(e); }
    else {  insertNE(d->right, e);  }
   } else  {
    if (d->left == NULL) {  d->left = makeSingleNode(e); }
    else {  insertNE(d->left, e);  }
   }
   return d;
}
```

/* Exercise: (i) Write an insert procedure to handle the case of the "empty tree". (ii) Modify insertNE to handle duplicates. End of Exercise. */

BST – Implementation of **Insert** [2]

```
OrdDict  makeSingleNode(Element e)
{
    OrdDict node;
    node = (OrdDict) malloc(sizeof(struct  __binTree));
    node->rootVal=e;
    node->left = node->right = NULL;
    return node;
}
```

BST: Implementation Of **Insert** - Duplicates [3]

```
//Preconditions:  d is non-empty;  keys are unique (i.e. duplicates);
OrdDict  insertNE(OrdDict d, Element e)
{
  if (d->rootVal.key < e.key)  {
    if (d->right == NULL) { d->right = makeSingleNode(e); }
    else {  insertNE(d->right, e);  }
   } else  {
    if (d->left == NULL) {  d->left = makeSingleNode(e); }
    else {  insertNE(d->left, e);  }
   }
   return d;
}
```

Exercise: Modify implementation for multiple elements with the same key (use one of the options):

- return success but do nothing,
- return failure with message “already found”,
- return success after adding new element separately,
- return success after overwriting contents.

End of Exercise

BST – Implementation of **Delete**

OrdDict delete(OrdDict dct, Key **k**)

- find the node, say **nd**, with contents matching key **k**
- if no such node exists done

  else if **nd** is a leaf then delete **nd** // must free nd

  else if one of the children of **nd** is empty

  then replace **nd** with the other subtree of **nd**

  else

  in-order successor of **nd** will : (i) be within the right subtree and (ii) have an empty left subtree

  a. find in-order successor of **nd**, say **suc**
  b. swap contents of **suc** with **nd**
  c. if **suc** is a leaf-node then delete **suc** // must free suc

     else replace **suc** with its right sub-tree

BST: Procedure Delete

- OrdDict delete(OrdDict dct, Key **k**)

```
{
  nd=dct; par=NULL;
  while (nd!=NULL) {
    if (nd->rootVal.key==k)  break;
    par=nd; nd=(nd->rootVal.key<k) ? nd->right : nd->left;
  }
  if (nd==NULL) return dct;  /* k not found */
  /* Postcondition : nd is the node to be deleted. */
  if (par==NULL) return deleteNE(nd);
  else {  // deleteNE will return a modified nd
        if (par->left==nd) par->left = deleteNE(nd);
        else /*par->right==nd */ par->right = deleteNE(nd);
        return dct;
  }
}
```

BST: PROCEDURE DELETE [2]

```
BinTree deleteNE(BinTree nd)
{ /*Precondition: nd contains the element to be deleted
    if (nd->right==NULL && nd->left==NULL) { free(nd); return NULL;
   } else if (nd->right==NULL) { temp=nd->left; free(nd); return temp;
   } else if (nd->left==NULL) { temp=nd->right; free(nd); return temp;
   } else {  par=nd; suc=nd->right;
            while (suc->left!=NULL) { par=suc; suc=suc->left; }
            /* Postcondition: suc points to in-order successor of nd */
            nd->rootVal = suc->rootVal;
            if (par->left==suc) { par->left=NULL; }
            else /* par->right==suc */ { par->right==NULL; }
            free(suc); return nd;
   }
}
```