Chapter 11-1, Lecture Notes

The ADT Table

1. A “table” would not be your favorite coffee table but rather something more along the lines of “A table of all major cities of the world”, like in figure 11-1 on page 590:

City	Country	Population
Athens	Greece	2,500,000
Barcelona	Spain	1,800,000
Cairo	Egypt	9,500,000
London	England	9,400,000
New York	U.S.A.	7,300,000
Paris	France	2,200,000
Rome	Italy	2,800,000
Toronto	Canada	3,200,000
Venice	Italy	300,000

1.1 The idea of a ADT Table operations would be:
- Insert a data item containing the value x
- Delete a data item containing the value x
- Ask a question about a data item containing the value x

1.2 For example:
- Find the phone number of John Smith
- Delete all the information about the employee with ID number 12908

1.3 The above table is ordered by city and so you could do a form of binary search on the name to find an item.

1.4 However, if you wanted to find all cities in Italy then you have no choice but to scan the entire table.

2. The ADT table (also called “dictionary”) allows you to look up information easily and has a specialized operation for this purpose.

2.1 The items in an ADT table are in form of records and you would use a field of the record as the specified search key.

2.1.1 For instance, it is natural to use the city name as a key in our example table.

2.1.2 The choice of a search key tells the ADT implementation:

Arrange the data to facilitate the search for an item, given the value of its search key.

2.2 The basic ADT table operations are as follows:

2.2.1 Create an empty table
2.2.2 Destroy a table
2.2.3 Determine whether a table is empty
2.2.4 Determine the number of items in a table
2.2.5 Insert a new item into a table
2.2.6 Delete the item with a given search key from a table
2.2.7 Retrieve the item with a given search key from a table
2.2.8 Traverse the items in a table in a sorted search-key order

3. For simplicity, we'll assume the values in the search key are unique and the ADT implementation will reject any duplicate keys.

   3.1 Operational Contract for the ADT Table

TableItemType is the type of the items stored in the table.

tableIsEmpty():boolean {query}
    Determines whether this table is empty.

tableLength():integer {query}
    Determines the number of items in this table.

tableInsert(in newItem:TableItemType) throw TableException
    Inserts newItem into this table whose items have distinct search keys that differ from newItem's search key. Throws TableException if the insertion is not successful.

tableDelete(in searchKey:KeyType) throw TableException
    Deletes from this table the item whose search key equals searchKey. If no such item exists, the method thows TableException

tableRetrieve(in searchKey:KeyType, out tableItem:TableItemType) {query}
    throw TableException
    Retrieves into tableItem the item in this table whose search key equals searchKey. If no such item exists, the method throw TableException.

traverseTable(in visit:FunctionType)
    Traverses this table in sorted search-key order and calls the function visit() once for each item.

   3.2 The above is not the end-all for operations.
      3.2.1 There may be additional ones needed or the old ones changed.
      3.2.2 You could also have a situation where there could be duplicate search keys in which you need more information to obtain the correct record.

   3.3 If you had only tableInsert, tableDelete, and tableRetrieve, then you cannot do:

Display all the table items

3.4 Because you cannot retrieve a data item without knowing the search key. T
  3.4.1 hat is why the traverseTable operation is important since it can visit each item in the table once.
  3.4.2 It has the "visit" parameter that is a function.
  3.4.3 The search key for the item should not be modifiable (the other fields can be but not search key).
  3.4.4 We will use the KeyedItem class from chapter 10 like so:

```
#include <string>

using namespace std;

typedef string KeyType;

class KeyedItem
{
public:
        KeyedItem() {}
        KeyedItem(const KeyType& keyValue) : searchKey(keyValue) {}
        KeyType getKey() const // returns search key
        {
                return searchKey;
        } / end getKey
private:
        KeyType searchKey;
}; // end KeyedItem
```

3.5 The only way to set the searchKey is via the constructor thus making the searchKey not modifiable.

4. Now if we have a class of City like so:

```
#include <string>

using namespace std;

class City
{
public:
        . . .
        // methods for access private data members
private:
        string cityName;        // city's name
        string country;         // city's country
        int pop;                // city's population
}; // end city
```

4.1 Now the operations you want to perform on the sample table would be:

- Display, in alphabetical order, the name of each city and its population
- Increase the population of each city by 10 percent
- Delete all cities with a population of less than 1,000,000

4.2 That would suggest using cityName as the search key.
4.3 So we can inherit from KeyedItem and get City to be:

```
class City : public KeyedItem
{
public:
        City() : KeyedItem() {}
        City(const string& name, const string& city, const int& num) :
                KeyedItem(name), country(ctry), pop(num) {}
        string cityName() const;
        int getPopulation() const;
        void setPopulation(int newPop);
private:
        // city's name is search-key value
        string country;         // city's country
        int pop;                // city's population
}; // end City
```

4.4 The first task would be to display items in alphabetic order.
  4.4.1 The traverseTable operation would have the displayItem function passed to it:

```
dsiplayItem(in anItem:TableItemType)

        Display anItem.cityName()
        Display anItem.getPopulation()
```

  4.4.2 For the next two operations, the order is immaterial.
  4.4.3 So traverseTable can use updatePopulation like so:

```
updatePopulation(inout anItem:TableItemType)

        anItem.setPopulation(1.1 * anItem.getPopulation())
```

  4.4.4 The third task you pass to traverseTable would be deleteSmall, like so:

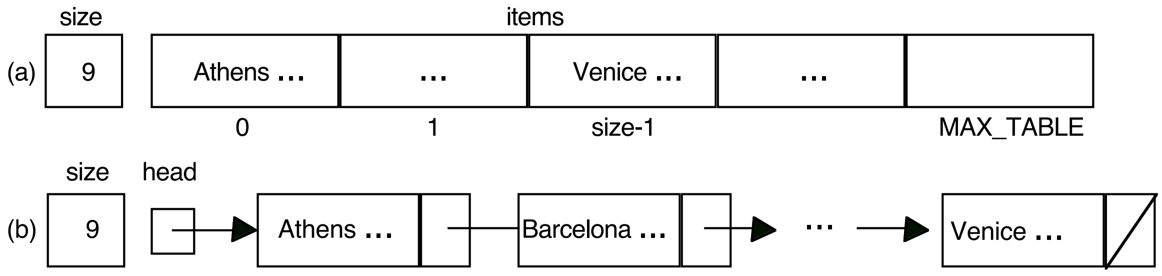
```
deleteSmall(inout t:Table, in anItem:TableItemType)

    t.tableDelete(anItem)
```

4.5 One problem with the deleteSmall is what happens when you delete an item and you must go to the next item.
  4.5.1 Could the traverseTable skip an item because of the delete?
  4.5.2 A difficult problem.

5. To implement the ADT table you could do one of the following:

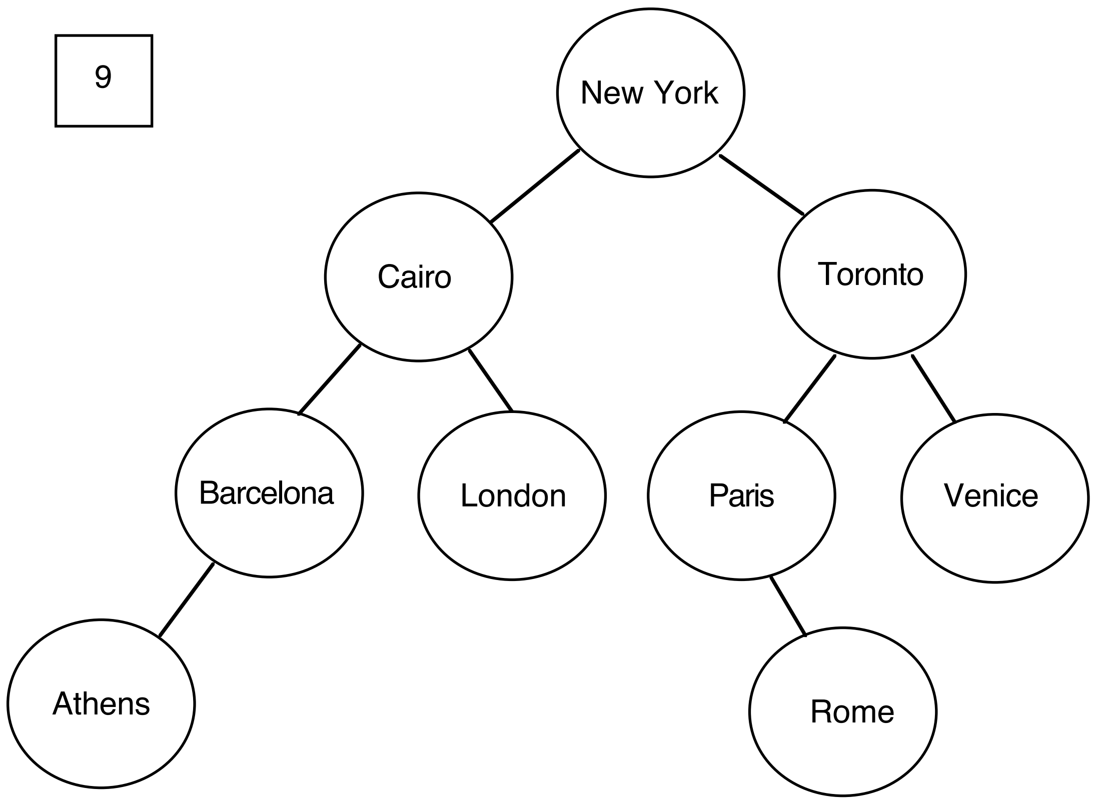
- Unsorted, array based
- Unsorted, pointer based
- Sorted (by search key), array based
- Sorted (by search key), pointer based

5.1 With unsorted, you can insert an item in any convenient.
  5.1.1 With sorted, you need to insert into a proper position.
  5.1.2 The figure 11-3, page 596 shows the array and pointer based implementations:



5.2 You cannot implement all operations for all applications.
  5.2.1 There is a trade-off for different applications.
  5.2.2 Some implementations are better for certain applications than others – some operations can be more easily done with a certain implementation.
  5.2.3 In deciding an implement, ask yourself if a certain operation would be done more frequently and thus favor an implementation.

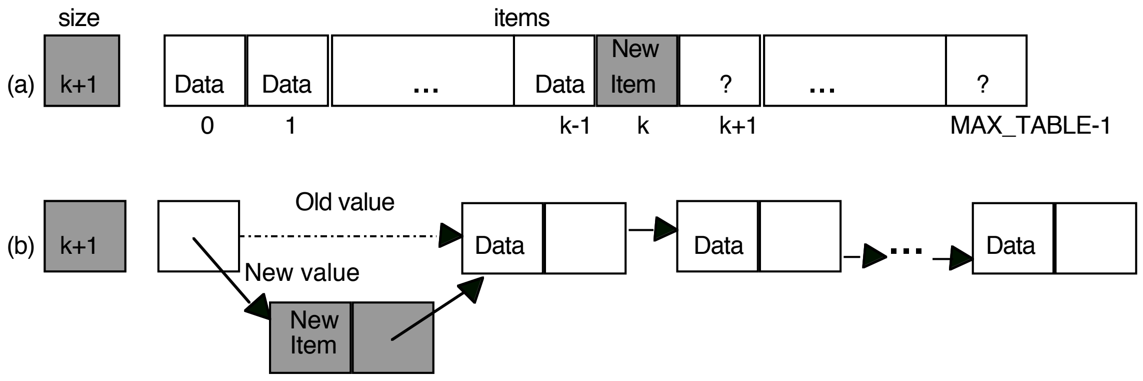
5.3 You can also use a binary search tree to implement a table.
  5.3.1 With a binary search tree implementation, you would have what is on page 597, figure 11-4:



5.3.2 Let's look at some scenarios.

5.3.3 Scenario A: insertion and traversal in no particular order.

5.3.3.1 Mary's sorority wants to raise money for a local charity.
5.3.3.2 They have a brainstorming session to discover new money-making strategies.
5.3.3.3 Mary records them by putting them in a table.
5.3.3.4 Order is not important – can be ordered or unordered.
5.3.3.5 The tableInsert can be efficient with unsorted.
5.3.3.6 With the array based implementation, all you need do is to insert at the end of the array and increment the size.
5.3.3.7 With pointer-based implementation, just insert at the beginning of the linked list.
5.3.3.8 Pointer-based is good if the number of items to be inserted is unknown.
5.3.3.9 Figure 11-5 on page 598 shows the implementations:



5.3.4 Scenario B: retrieval.
- 5.3.4.1 With a word processor's thesaurus to look up synonyms for a word, you use a retrieval operation.
- 5.3.4.2 The ADT table that represents the thesaurus would have the word and the synonym.
- 5.3.4.3 Frequent retrieval operations needs an efficient algorithm.
- 5.3.4.4 Typically you do not alter the thesaurus so insert and deletion is not necessary.
- 5.3.4.5 The data needs to be sorted if you are using a binary search with array-based implementation.
- 5.3.4.6 The linked-list implementation has other issues.
- 5.3.4.7 The two questions you need to ask is:
- 5.3.4.8 Is a binary search of a linked list possible?
- 5.3.4.9 How much more efficient is a binary search of an array than a sequential search of a linked list?

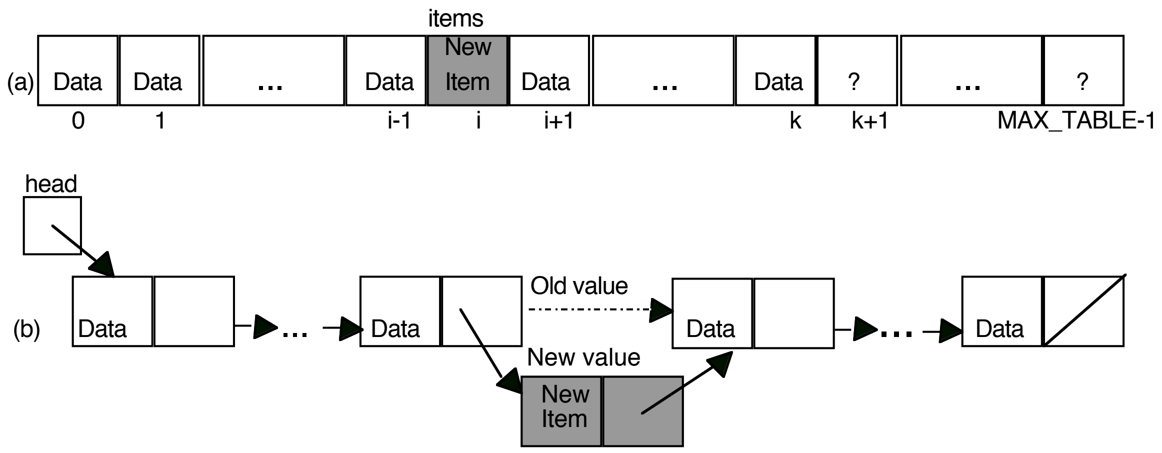
5.3.5 The binary search starts out by looking in the middle of the list.
- 5.3.5.1 With a linked list, that is difficult to do.
- 5.3.5.2 The problem is compounded with the recursive call where you need to find the middle of the next half f the list.
- 5.3.5.3 But with an array, accessing the middle is very easy and quick.
- 5.3.5.4 The binary search has $O(\log_2 n)$ efficiency which for a large set of words – an array based implement would be the obvious choice for a thesaurus.

5.3.6 Scenario C: insertion, deletion, retrieval, and traversal in sorted order.
- 5.3.6.1 The local library has computerized their catalog of books.
- 5.3.6.2 You perform a retrieval when you access the catalog.
- 5.3.6.3 The staff used insertion and deletion to update the catalog.
- 5.3.6.4 While inserting and deleting are not done frequently, they cannot be ignored nor can they be inefficient.
- 5.3.6.5 The basic operations for insert and delete are:

  - Find the appropriate position in the table.
  - Insert into (or delete from) this position.

5.3.7 Step 1 is more efficient if the binary search is used – so the array-based implementation is better.

5.3.7.1 However, for step 2, the pointer-based implementation would be best. The problem with array-based, the items have to be moved for both insert and delete. Figure 11-6 on page 601 shows the implementations:



6 You could also implement an ADT table using a binary tree.

6.1 You would need to keep the height of the tree close to minimum – that can be difficult with many insertions and deletions.

6.2 Figure 11-7 on page 602 has the efficiencies of the operations for a sorted array-based implementation:

	Insertion	Deletion	Retrieval	Traversal
Unsorted array based	O(1)	O(n)	O(n)	O(n)
Unsorted pointer based	O(1)	O(n)	O(n)	O(n)
Sorted array based	O(n)	O(n)	O(log n)	O(n)
Sorted pointer based	O(n)	O(n)	O(n)	O(n)
Binary search tree	O(log n)	O(log n)	O(log n)	O(n)

6.3 There are three reasons to pick a particular implement.

6.3.1 First is perspective – and not overanalyzing a problem.

6.3.1.1 If the size of the problem is small then the differences of the implementations are not significant.

6.3.1.2 An array-based would be sufficient.

6.3.2 Second reason is efficiency.

6.3.2.1 A linear implementation can be quite efficient for certain situations – like scenario A where insertion and traversal are important.

6.3.2.2 Scenario B would need a sorted array-based implement since retrieval is important.

6.3.3 Third reason is motivation.

6.3.3.1 Linear implement could be inadequate therefore looking at the binary search tree would make sense.

7 There are implementation for the sorted array of ADT Table is in Documents/Table/SortedArray. The implementation for the binary tree of ADT Table is in Documents/Table/BinaryTree.