NOTTINGHAM TRENT UNIVERSITY

Software design and implementation 2

Pedro Machado pedro.baptistamachado@ntu.ac.uk

Overview

- Searching Algorithms
- Sorting Algorithms

Sorting & Searching

Fundamental problems in computer science and programming

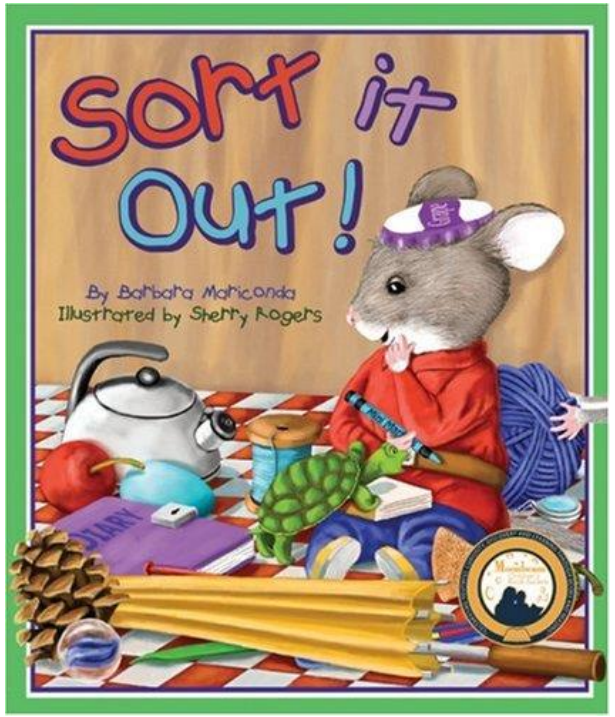
Sorting done to make searching easier

Multiple different algorithms to solve the same problem

- How do we know which algorithm is "better"?

Look at searching first

Examples will use arrays of ints to illustrate algorithms



Searching

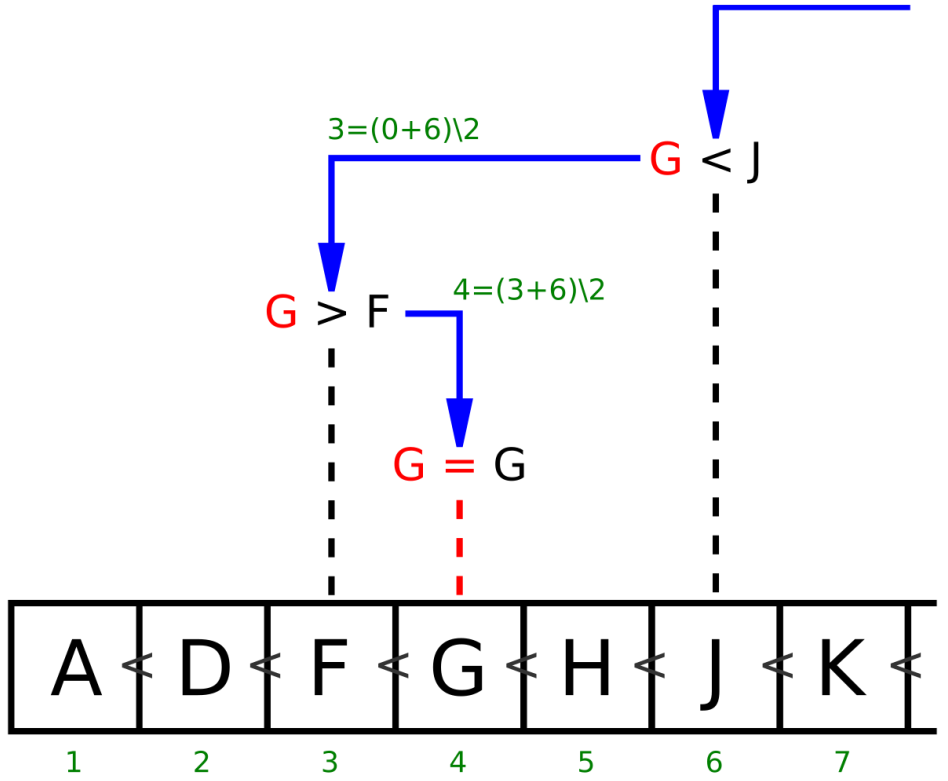
- Given a list of data find the location of a particular value or report that value is not present
- linear search
  - intuitive approach
  - start at first item
  - is it the one I am looking for?
  - if not go to next item
  - repeat until found or all items checked
- If items not sorted or unsortable this approach is necessary

Linear Search

```
/*     pre: list != null
       post: return the index of the first occurrence
       of target in list or -1 if target not present in
       list
*/
public int linearSearch(int[] list, int target) {
    for(int i = 0; i < list.length; i++)
        if( list[i] == target )
            return i;
    return -1;
}
```

Searching a sorted list

- If items are sorted then we can divide and conquer
- dividing your work in half with each step
  - generally a good thing
- The Binary Search on List in Ascending order
  - Start at middle of list
  - is that the item?
  - If not is it less than or greater than the item?
  - less than, move to second half of list
  - greater than, move to first half of list
  - repeat until found or sub list size = 0



Binary Search - Generic

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53

```
public static int bsearch(int[] list, int target)
{       int result = -1;
        int low = 0;
        int high = list.length - 1;
        int mid;
        while( result == -1 && low <= high )
        {       mid = low + ((high - low) / 2);
                if( list[mid] == target )
                        result = mid;
                else if( list[mid] < target)
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        return result;
}
// mid = ( low + high ) / 2; // may overflow!!!
// or mid = (low + high) >>> 1; using bitwise op
```

Binary Search - Recursive

```
public static int bsearch(int[] list, int target){
    return bsearch(list, target, 0, list.length - 1);
}

public static int bsearch(int[] list, int target,
                          int first, int last){
    if( first <= last ){
        int mid = low + ((high - low) / 2);
        if( list[mid] == target )
            return mid;
        else if( list[mid] > target )
            return bsearch(list, target, first, mid - 1);
        else
            return bsearch(list, target, mid + 1, last);
    }
    return -1;
}
```

Binary Search Tree

Worst case complexity?

What is the maximum depth of recursive calls in binary search as function of n?

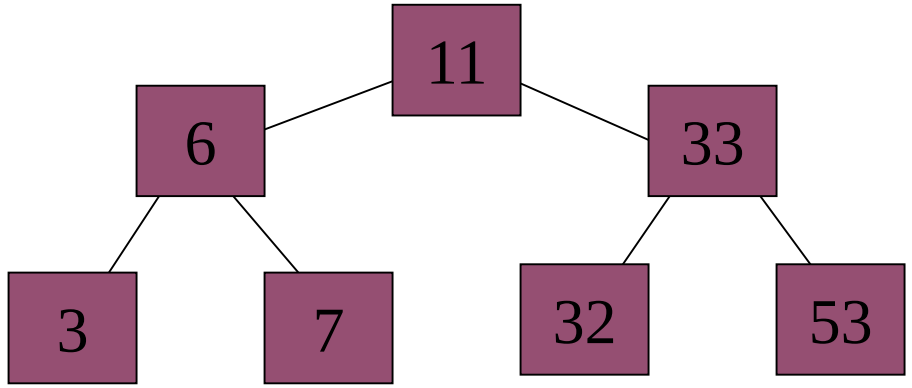
Each level in the recursion, we split the array in half (divide by two).

Therefore maximum recursion depth is floor(log2n) and worst case = O(log2n).

Average case is also = O(log2n).

Example:

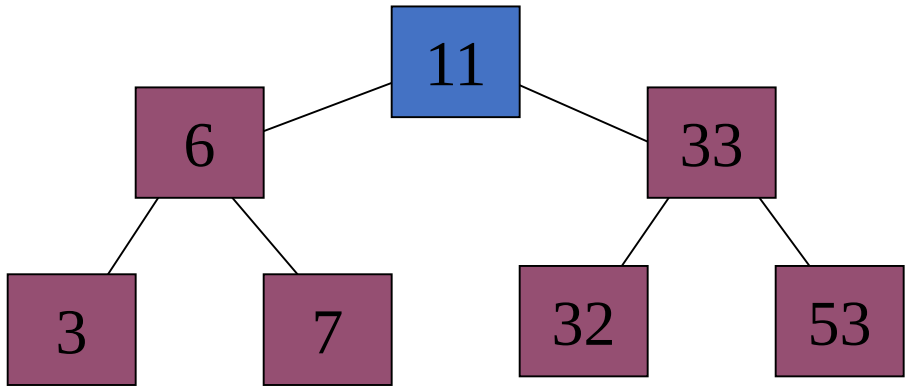
Corresponding Binary search tree



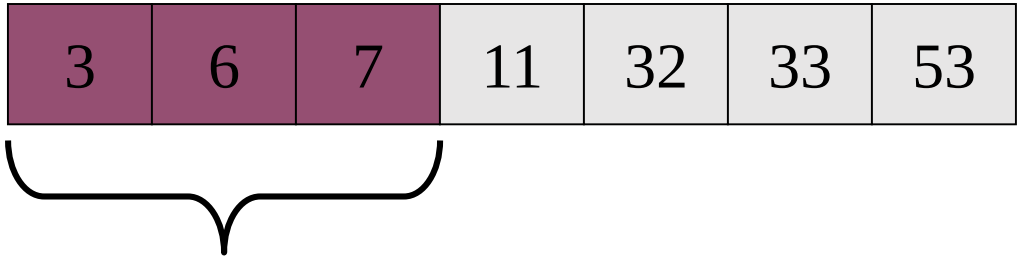
Search for target = 7

Find midpoint:

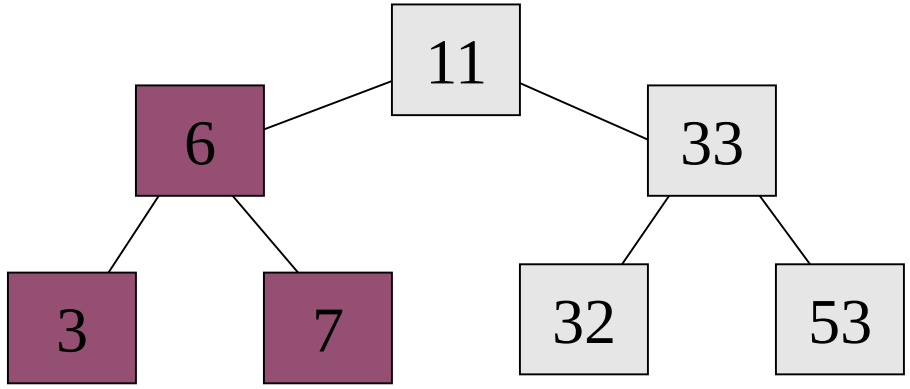
Start at root:



Search left subarray:



Search left subtree:

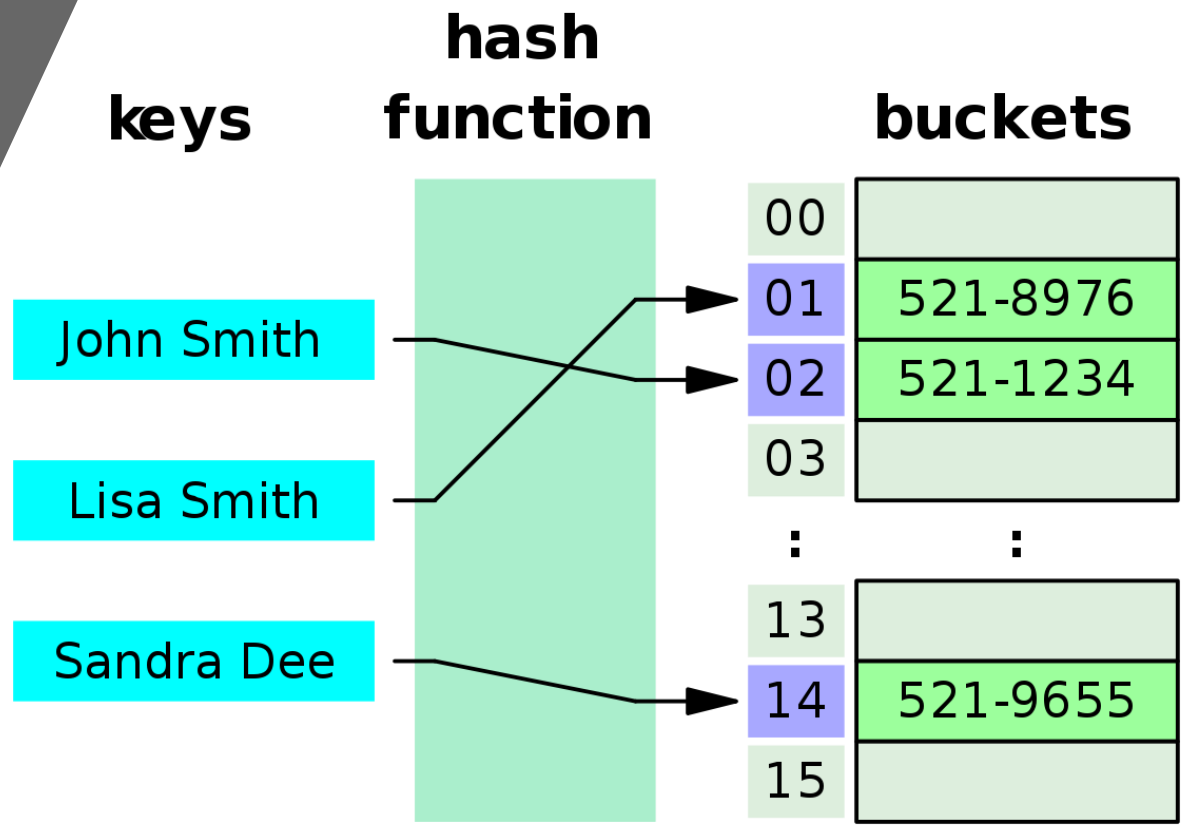


Can we do better than O(log2n)?

- Average and worst case of serial search = O(n)
- Average and worst case of binary search = O(log2n)

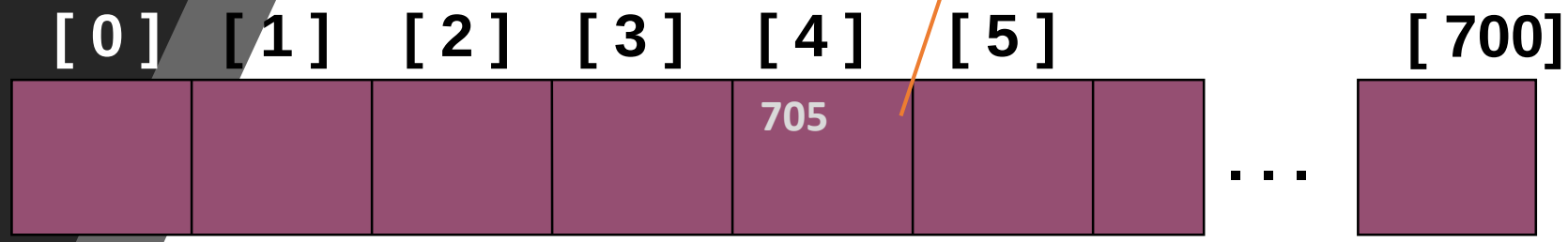
- Can we do better than this?

YES. Use a hash table!



What is a Hash Table ?

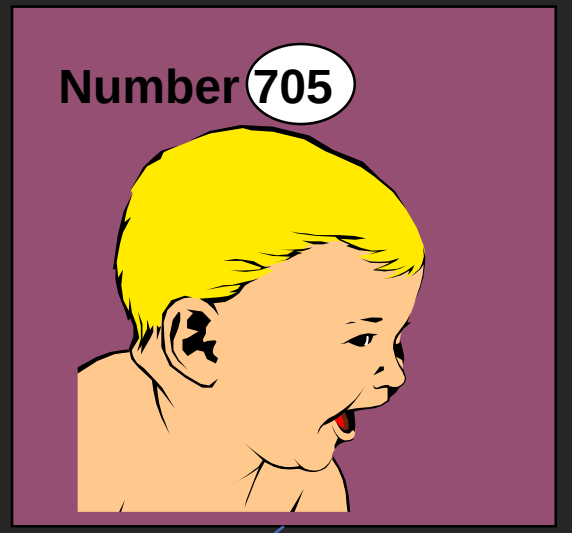
- The simplest kind of hash table is an array of records.
- This example has 701 records.
- Each record has a special field, called its **key**.
- In this example, the **key** is a long integer field called Number.
- The number might be a person's identification number, and the rest of the record has information about the person.

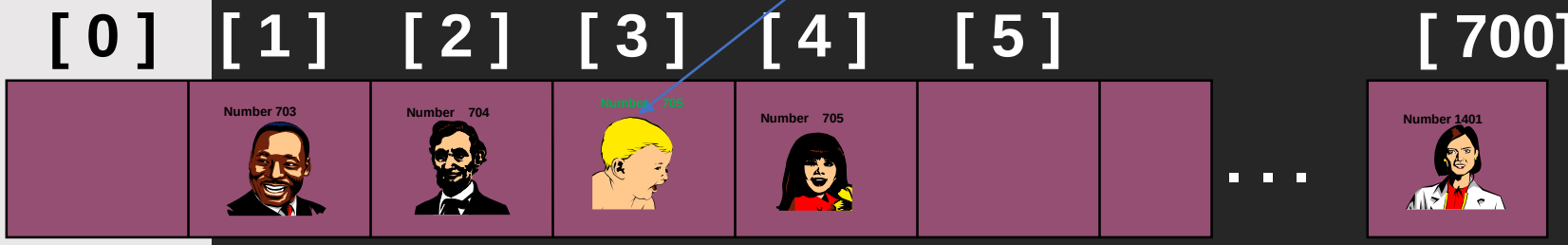


What is a Hash Table?

When a hash table is in use, some spots contain valid records, and other spots are "empty".

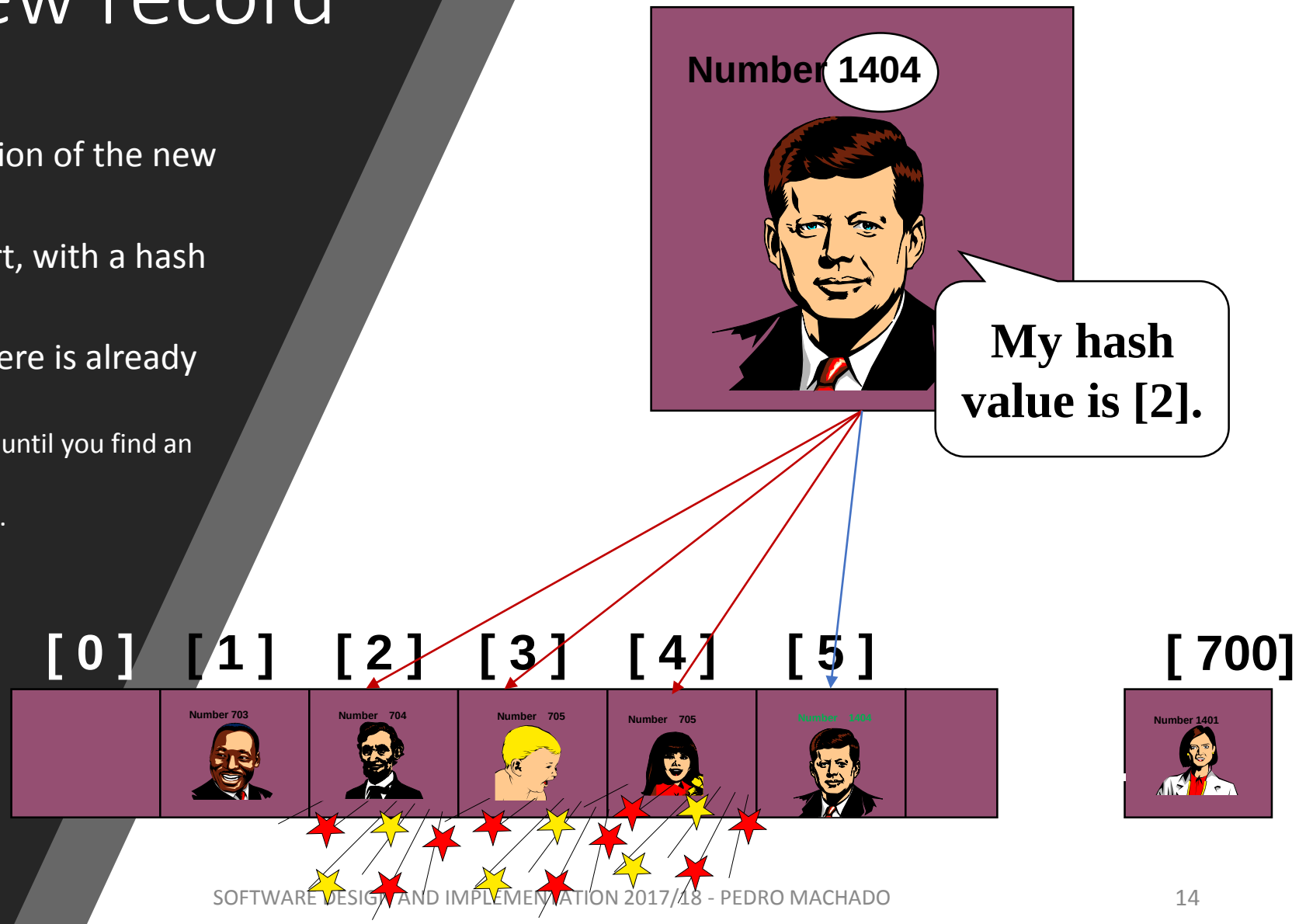
- In order to insert a new record, the **key** must somehow be **converted to** an array index.
- The index is called the **hash** value of the key.
- Typical way create a hash value:
- Number mod length(matrix) or (e.g. 704 mod 701 = 3)





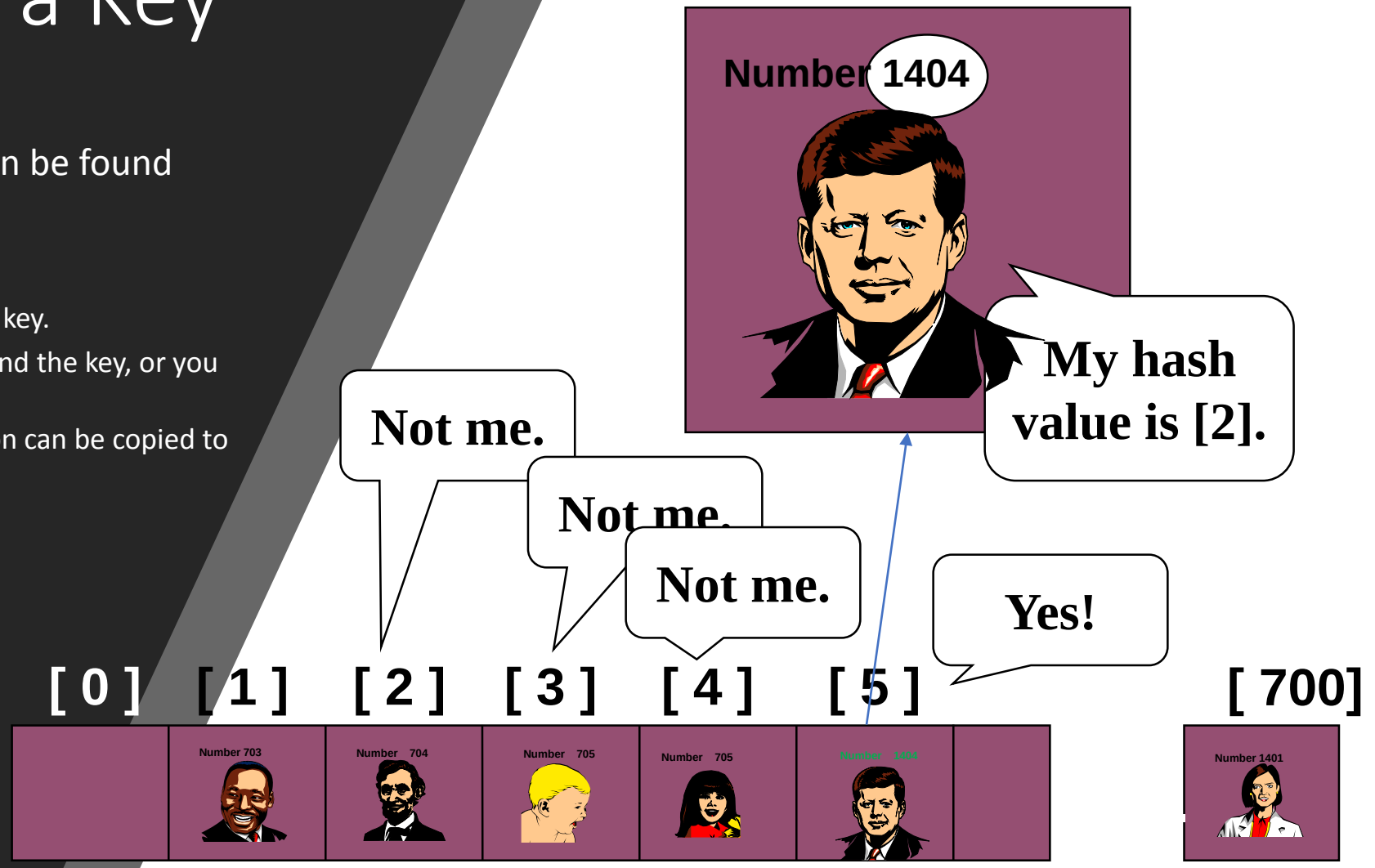
Inserting a new record

- The hash value is used for the location of the new record
- Here is another new record to insert, with a hash value of 2.
- This is called a collision, because there is already another valid record at [2].
  - When a collision occurs, move forward until you find an empty spot.
  - The new record goes in the empty spot.



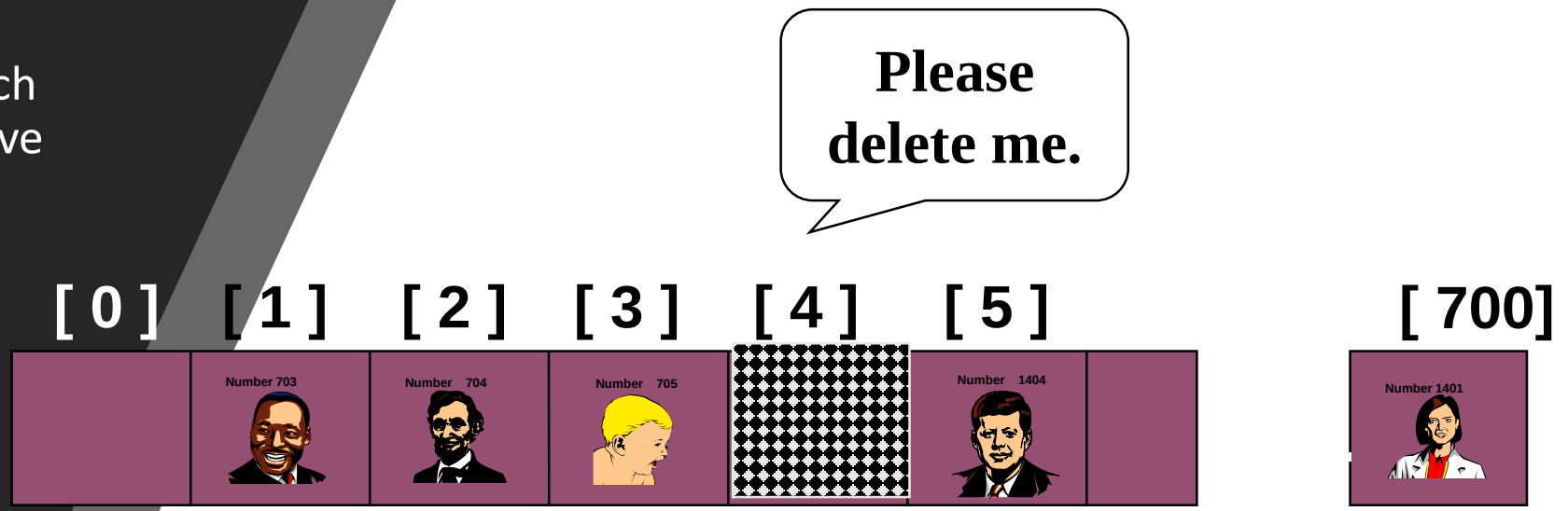
Searching for a Key

- The data that's attached to a key can be found fairly quickly.
- Calculate the hash value.
  - Check that location of the array for the key.
  - If not, keep moving forward until you find the key, or you reach an empty spot.
  - When the item is found, the information can be copied to the necessary location.



Deleting a Record

- Records may also be deleted from a hash table.
- But the location must not be left as an ordinary "empty spot" since that could interfere with searches.
- The location must be marked in some special way so that a search can tell that the spot used to have something in it.



Hashing

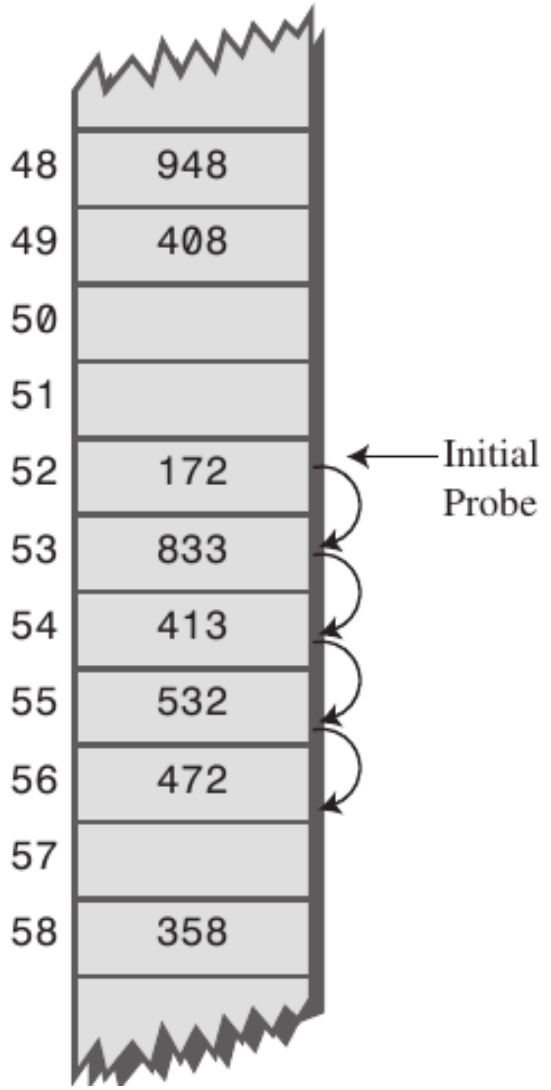
- Hash tables store a collection of records with keys.
- The location of a record depends on the hash value of the record's key.
- Open address hashing:
- When a collision occurs, the next available location is used.
- Searching for a particular key is generally quick.
- When an item is deleted, the location must be marked in a special way, so that the searches know that the spot used to be used.

To reduce collisions…

- Use table CAPACITY = prime number of form 4k+3
- Hashing functions:
  - Division hash function: key % CAPACITY
  - Mid-square function: (key*key) % CAPACITY
  - Multiplicative hash function: key is multiplied by positive constant less than one. Hash function returns first few digits of fractional result.

Clustering

- In the hash method described, when the insertion encounters a collision, we move forward in the table until a vacant spot is found. This is called linear probing.
- Problem: when several different keys are hashed to the same location, adjacent spots in the table will be filled. This leads to the problem of clustering.
- As the table approaches its capacity, these clusters tend to merge. This causes insertion to take a long time (due to linear probing to find vacant spot).

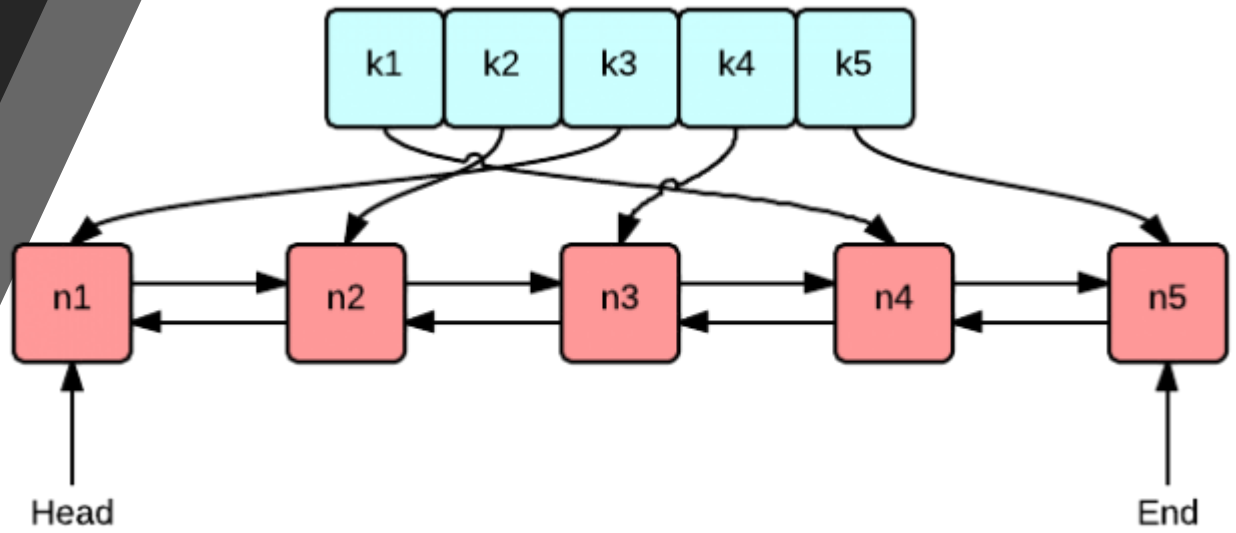


Double Hashing

- One common technique to avoid cluster is called double hashing.
- Let's call the original hash function hash1
- Define a second hash function hash2

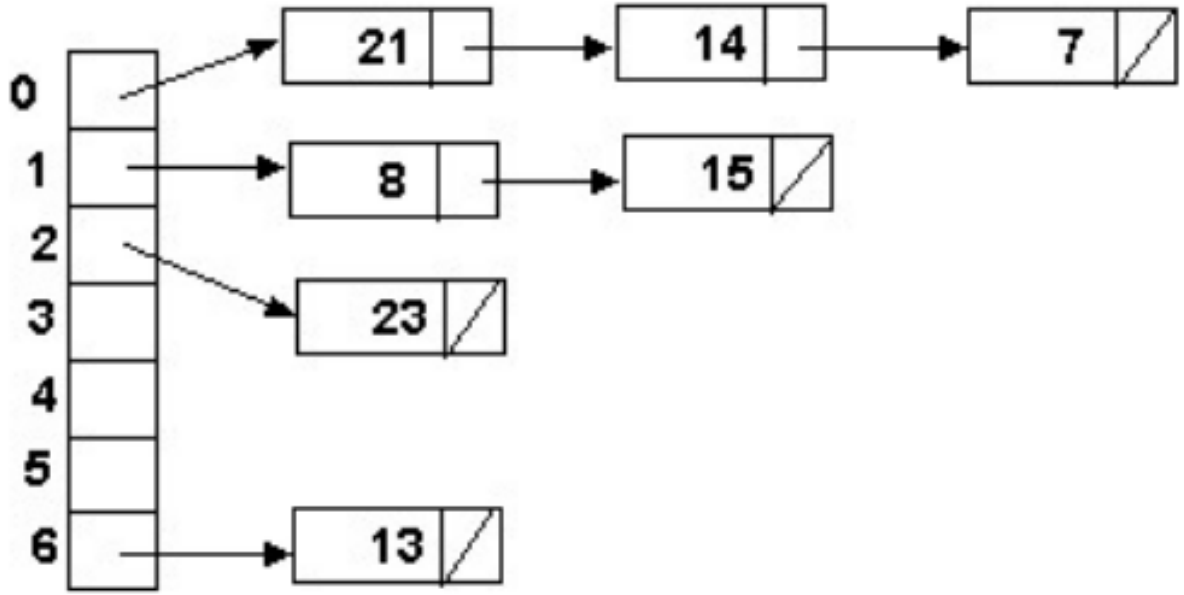
- Double hashing algorithm:
- When an item is inserted, use hash1(key) to determine insertion location i in array as before.
- If collision occurs, use hash2(key) to determine how far to move forward in the array looking for a vacant spot:

- next location = (i + hash2(key)) % CAPACITY



Chained Hashing

- In open address hashing, a collision is handled by probing the array for the next vacant spot.
- When the array is full, no new items can be added.
- We can solve this by resizing the table.
- Alternative: chained hashing.



Simple implementation

```
class HashEntry {
private:
	int key;
	int value;
public:
	HashEntry(int key, int value) {
		this->key = key;
		this->value = value;
	}

	int getKey() {
		return key;
	}

	int getValue() {
		return value;
	}
};
```

Simple implementation

```
const int TABLE_SIZE = 128;

class HashMap {
private:
	HashEntry **table;
public:
	HashMap() {
		table = new HashEntry*[TABLE_SIZE];
		for (int i = 0; i < TABLE_SIZE; i++)
			table[i] = NULL;
	}
```

Simple implementation

```
int get(int key) {
            int hash = (key % TABLE_SIZE);
            while (table[hash] != NULL &&
table[hash]->getKey() != key)
                  hash = (hash + 1) % TABLE_SIZE;
            if (table[hash] == NULL)
                  return -1;
            else
                  return table[hash]->getValue();
      }
```

Simple implementation

```
void put(int key, int value) {
            int hash = (key % TABLE_SIZE);
            while (table[hash] != NULL &&
table[hash]->getKey() != key)
                  hash = (hash + 1) % TABLE_SIZE;
            if (table[hash] != NULL)
                  delete table[hash];
            table[hash] = new HashEntry(key,
value);
      }

      ~HashMap() {
            for (int i = 0; i < TABLE_SIZE; i++)
                  if (table[i] != NULL)
                        delete table[i];
            delete[] table;
      }
};
```

Time Analysis of Hashing

- Worst case: every key gets hashed to same array index! O(n) search!!
- Luckily, average case is more promising.
- First we define a fraction called the hash table *load factor:*

$$\alpha = \frac{\textit{number of occupied table locations}}{\textit{size of table's array}}$$

Summary

- Serial search: average case O(n)
- Binary search: average case $O(\log_2 n)$
- Hashing
  - Open address hashing
    - Linear probing
    - Double hashing
  - Chained hashing
  - Average number of elements examined is function of load factor α.

Searching in graphs

- Traversal

Given G=(V,E) and vertex v, find all wεV, such that w connects v.

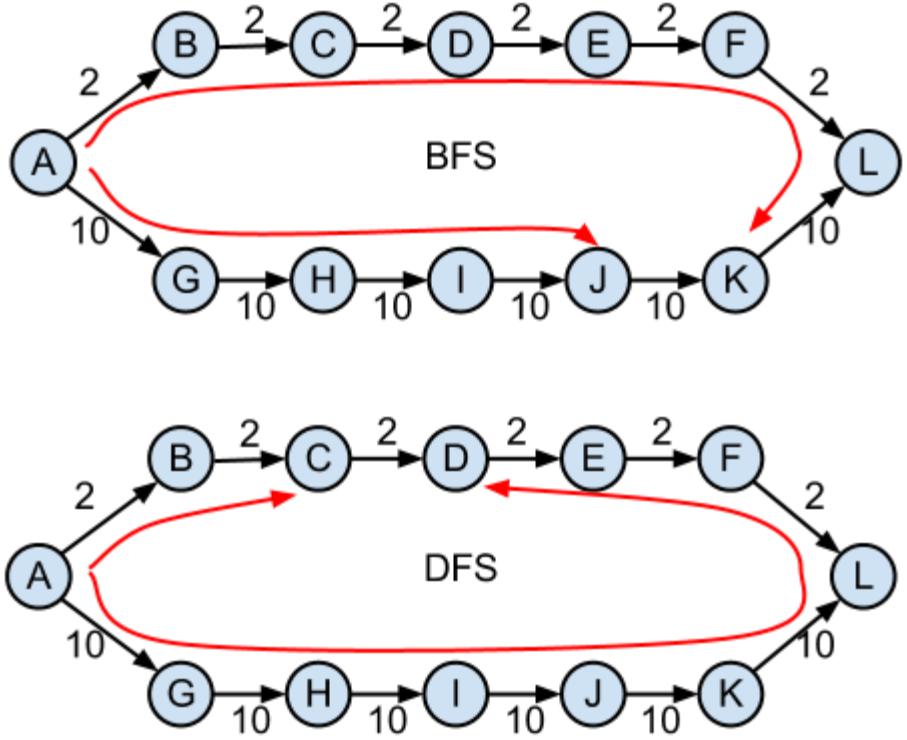
- Depth First Search (DFS)

pre-order tree traversal

- Breadth First Search (BFS)

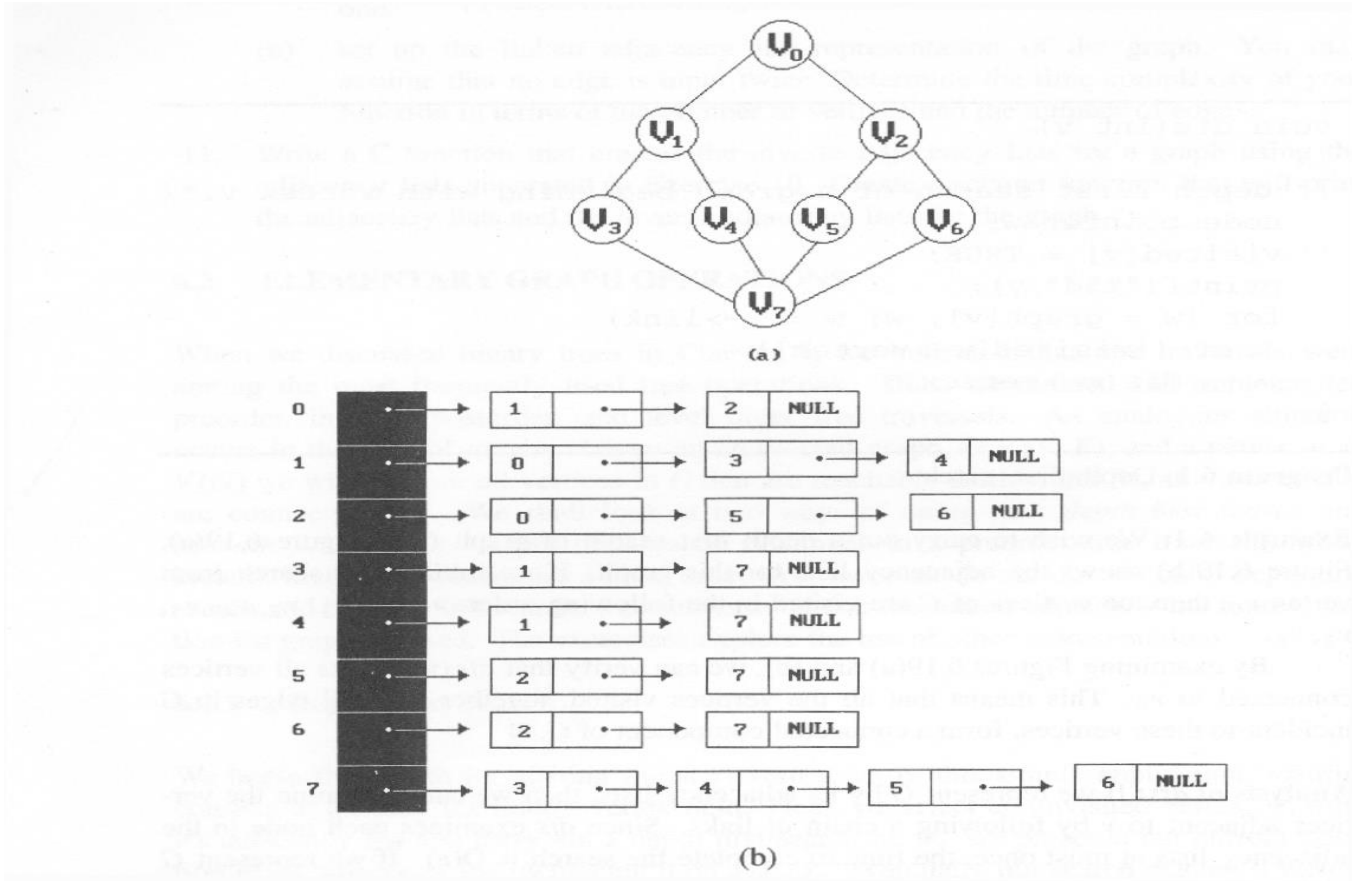
level order tree traversal

- Connected Components
- Spanning Trees



Depth First Search

depth first search: v0, v1, v3, v7, v4, v5, v2, v6



breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  cout << v << endln;
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Data structure
adjacency list: O(e)
adjacency matrix: $O(n^2)$

Breadth First Search

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *, queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  cout << v << endln;
  visited[v] = TRUE;
  addq(&front, &rear, v);
while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
        cout<< w->vertex << endln;
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

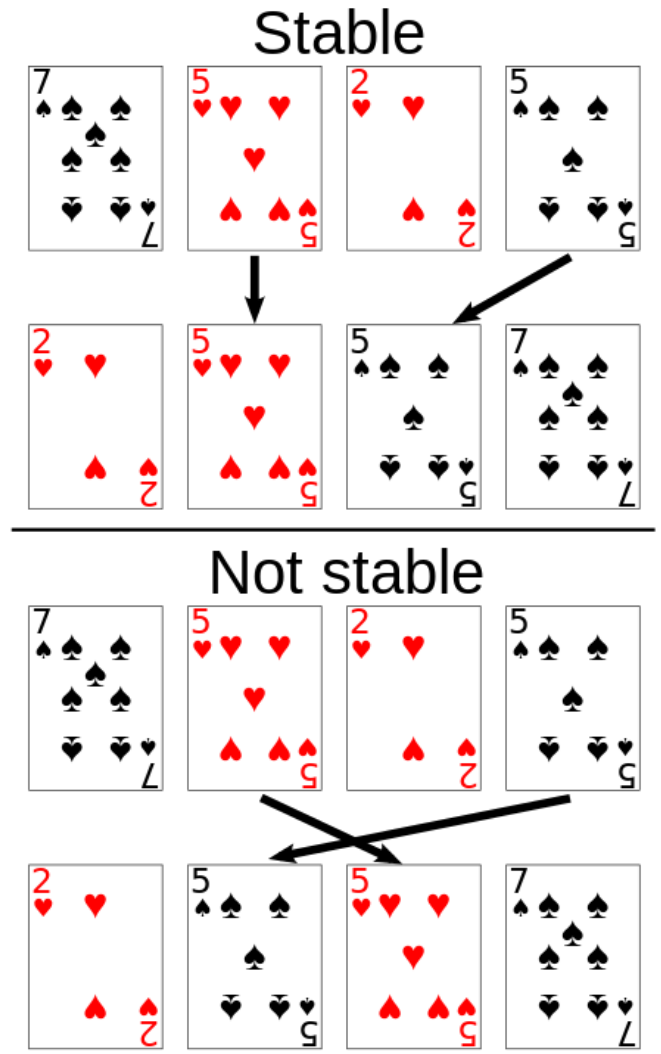
adjacency list: O(e)
adjacency matrix: $O(n^2)$

Sorting

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
- One of the difficulties with sorting is working with a fixed size storage container (array)
- if resize, that is expensive (slow)
- The "simple" sorts run in quadratic time $O(N^2)$
- bubble sort
- selection sort
- insertion sort

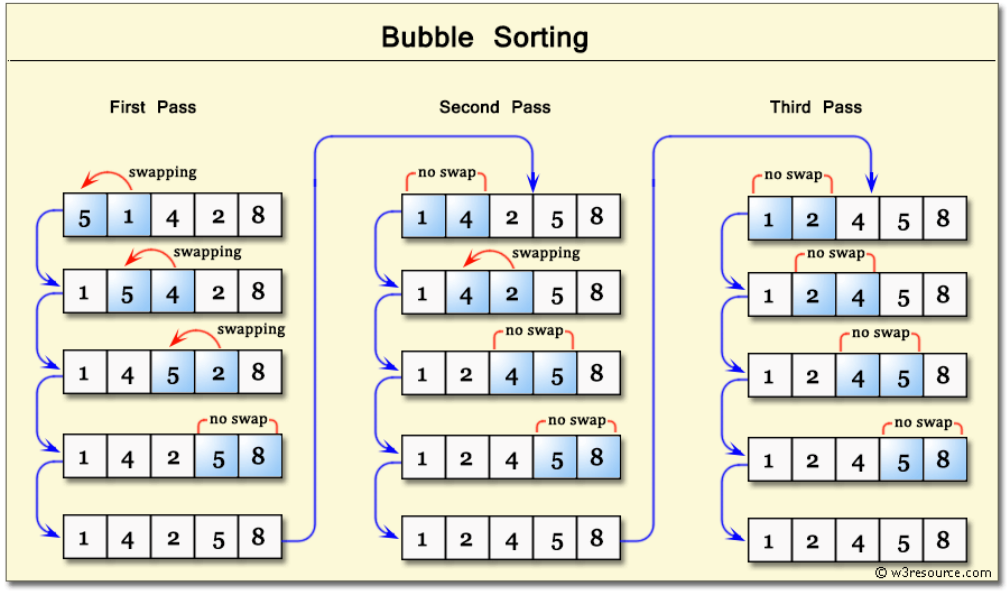
Stable Sorting

- A property of sorts
- If a sort guarantees the relative order of equal items stays the same then it is a stable sort
- [71, 6, 72, 5, 1, 2, 73, -5]
- subscripts added for clarity
- [-5, 1, 2, 5, 6, 71, 72, 73]
- result of stable sort
- Real world example:
- sort a table in Wikipedia by one criteria, then another
- sort by country, then by major wins



Bubble Sort

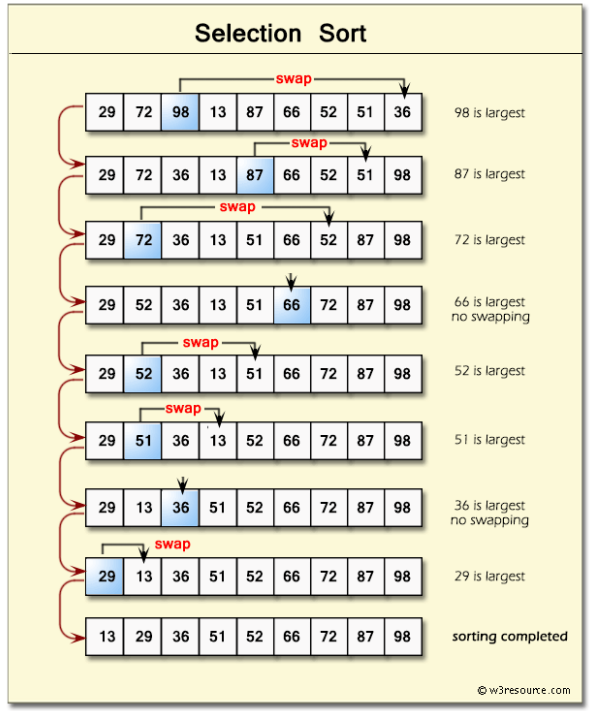
- One of the simplest sorting algorithms proceeds by walking down the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is continued until the list is sorted.




```
void BubbleSort(int List[] , int Size) {
   int tempInt;
   for (int Stop = Size - 1; Stop > 0; Stop--) {
      for (int Check = 0; Check < Stop; Check++) {
         if (List[Check] > List[Check + 1]) {
            tempInt          = List[Check];
            List[Check]      = List[Check + 1]; //   wrong orde
            List[Check + 1] = tempInt;
         }
      }
   }
}
```

Selection Sort

- Algorithm
- Search through the list and find the smallest element
- swap the smallest element with the first element
- repeat starting at second element and find the second smallest element



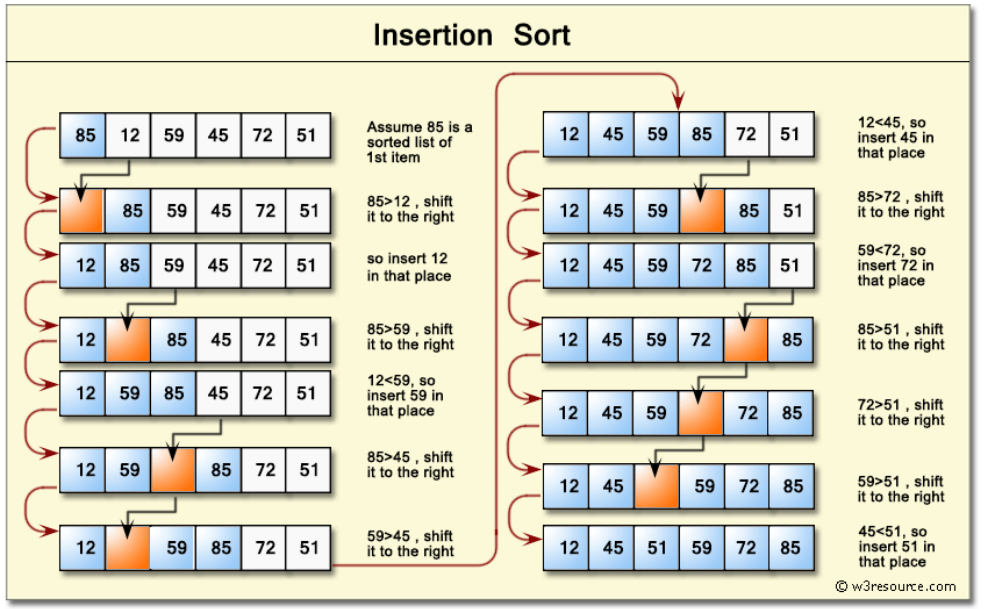

```
public static void selectionSort(int[] list){
        int min;
        int temp;
        for(int i = 0; i < list.length - 1; i++) {
                min = i;
                for(int j = i + 1; j < list.length; j++)
                        if( list[j] < list[min] )
                                min = j;
                        temp = list[i];
                        list[i] = list[min];
                        list[min] = temp;
                 }
        }
```

Generic Selection Sort

```
public void selectionSort(Comparable[] list)
   {  int min; Comparable temp;
      for(int i = 0; i < list.length - 1; i++) {
      {  min = i;
       for(int j = i + 1; j < list.length; j++)
      if( list[min].compareTo(list[j]) > 0 )
             min = j;
       temp = list[i];
       list[i] = list[min];
       list[min] = temp;
      }
   }
```

Insertion Sort

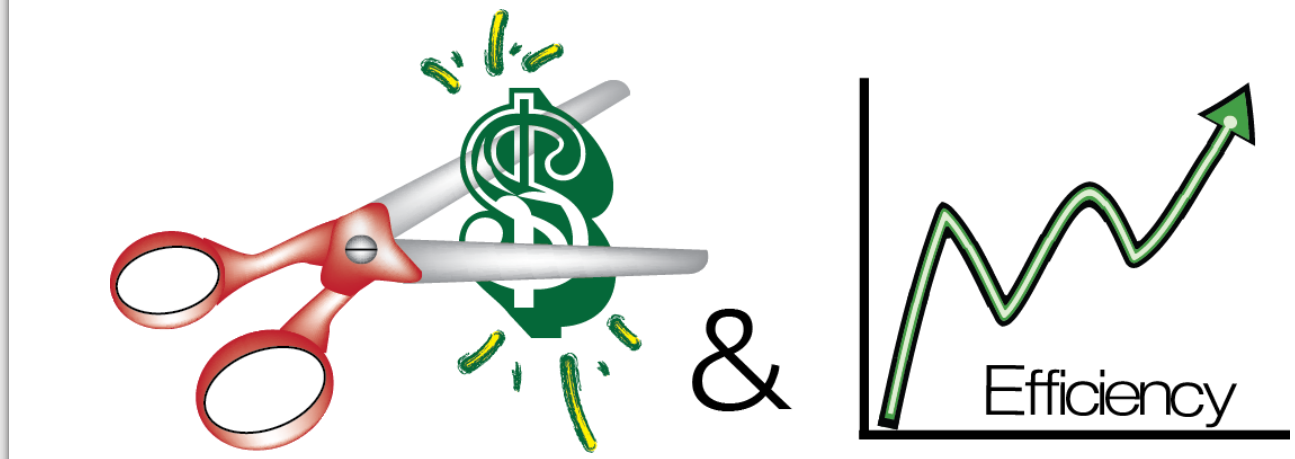
- Another of the O(N^2) sorts
- The first item is sorted
- Compare the second item to the first
- if smaller swap
- Third item, compare to item next to it
- need to swap
- after swap compare again
- And so forth…




```
public void insertionSort(int[] list)
{       int temp, j;
        for(int i = 1; i < list.length; i++)
        {       temp = list[i];
                j = i;
                while( j > 0 && temp < list[j - 1])
                {       // swap elements
                        list[j] = list[j - 1];
                        list[j - 1] = temp;
                        j--;
                }
        }
}
```

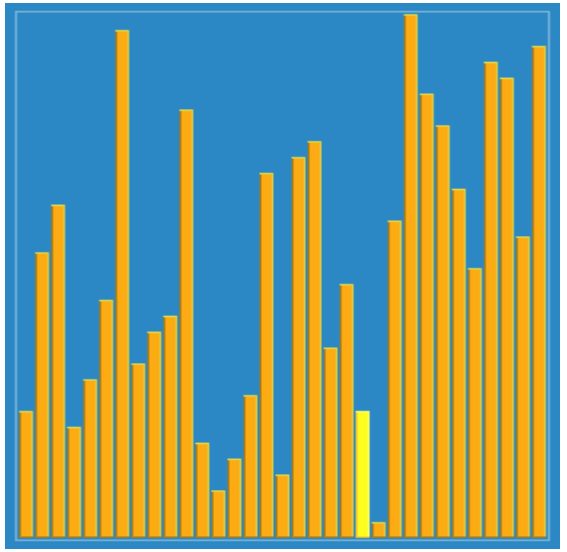
Sub-quadratic Sorting Algorithms

- Sub Quadratic means having a Big O better than O(N2)



Shell Sort

- Created by Donald Shell in 1959
- Wanted to stop moving data small distances (in the case of insertion sort and bubble sort) and stop making swaps that are not helpful (in the case of selection sort)
- Start with sub arrays created by looking at data that is far apart and then reduce the gap size



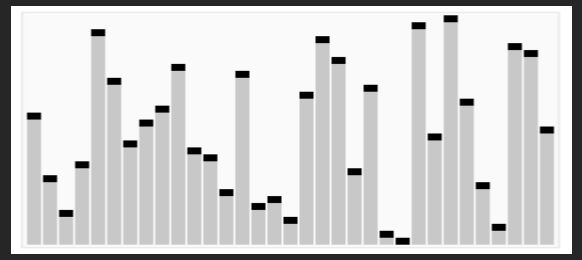
```
public static void shellsort(Comparable[] list)
{  Comparable temp; boolean swap;
   for(int gap = list.length / 2; gap > 0; gap /= 2)
      for(int i = gap; i < list.length; i++)
      {   Comparable tmp = list[i];
          int j = i;
          for( ; j >= gap &&
                          tmp.compareTo( list[j - gap] ) < 0;
                          j -= gap )
                list[ j ] = list[ j - gap ];
          list[ j ] = tmp;
       }
}
```

Quick Sort

- Invented by C.A.R. (Tony) Hoare
- A divide and conquer approach

that uses recursion

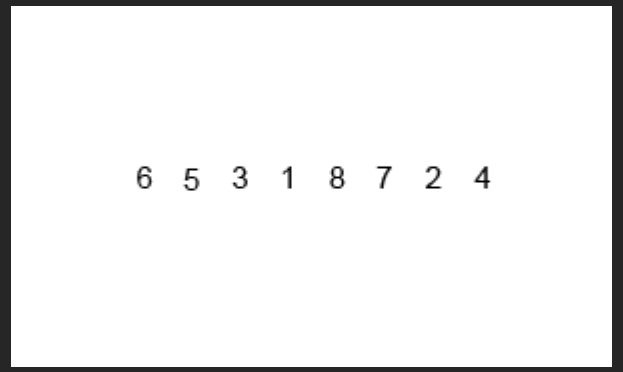
1. If the list has 0 or 1 elements it is sorted
2. otherwise, pick any element p in the list. This is called the pivot value
3. Partition the list minus the pivot into two sub lists according to values less than or greater than the pivot. (equal values go to either)
4. return the quicksort of the first list followed by the quicksort of the second list



```
public static void swapReferences( Object[] a, int index1, int index2
)
    {   Object tmp = a[index1];
        a[index1] = a[index2];
        a[index2] = tmp;
    }

     public void quicksort( Comparable[] list, int start, int stop )
     {    if(start >= stop)
            return; //base case list of 0 or 1 elements

           int pivotIndex = (start + stop) / 2;

           // Place pivot at start position
          swapReferences(list, pivotIndex, start);
          Comparable pivot = list[start];

          // Begin partitioning
          int i, j = start;

           // from first to j are elements less than or equal to pivot
           // from j to i are elements greater than pivot
           // elements beyond i have not been checked yet
           for(i = start + 1; i <= stop; i++ )
           {   //is current element less than or equal to pivot
               if(list[i].compareTo(pivot) <= 0)
               {   // if so move it to the less than or equal portion
                   j++;
                   swapReferences(list, i, j);
               }
           }

           //restore pivot to correct spot
           swapReferences(list, start, j);
           quicksort( list, start, j - 1 );    // Sort small elements
           quicksort( list, j + 1, stop );   // Sort large elements
       }
```

Merge Sort

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together




```
public static void mergeSort(Comparable[] c)
{           Comparable[] temp = new Comparable[ c.length ];
            sort(c, temp, 0, c.length - 1);
}

private static void sort(Comparable[] list, Comparable[] temp,
                                    int low, int high)
{           if( low < high){
                        int center = (low + high) / 2;
                        sort(list, temp, low, center);
                        sort(list, temp, center + 1, high);
                        merge(list, temp, low, center + 1, high);
            }
}
private static void merge( Comparable[] list, Comparable[] temp,
                                    int leftPos, int rightPos, int rightEnd){
            int leftEnd = rightPos - 1;
            int tempPos = leftPos;
            int numElements = rightEnd - leftPos + 1;
            //main loop
            while( leftPos <= leftEnd && rightPos <= rightEnd){
                        if( list[ leftPos ].compareTo(list[rightPos]) <= 0){
                                    temp[ tempPos ] = list[ leftPos ];
                                    leftPos++;
                        }
                        else{
                                    temp[ tempPos ] = list[ rightPos ];
                                    rightPos++;
                        }
                        tempPos++;
            }
            //copy rest of left half
            while( leftPos <= leftEnd){
                        temp[ tempPos ] = list[ leftPos ];
                        tempPos++;
                        leftPos++;
            }
            //copy rest of right half
            while( rightPos <= rightEnd){
                        temp[ tempPos ] = list[ rightPos ];
                        tempPos++;
                        rightPos++;
            }
            //Copy temp back into list
            for(int i = 0; i < numElements; i++, rightEnd--)
                        list[ rightEnd ] = temp[ rightEnd ];
}
```

Question

What sorting algorithm to choose?

Tip: The following link will help answering the question.

https://www.toptal.com/developers/sorting-algorithms

Summary

- Searching Algorithms
- Sorting Algorithms