Arrays, Vectors
Searching, Sorting

Arrays

- `char s[200];` //array of 200 characters
  - different type than class `string`
- can be accessed as `s[0], s[1], ..., s[199]`
  - `s[0]='H'; s[1]='e'; s[2]='l'; s[3]='l'; s[4]='o';`
  - `char a = s[3];`
- works for any type
  - `double d[10]; int i[100];`
- <u>C++ does not check for array bounds !!</u>

Memory allocation for arrays

`int A[n]//allocates n*size(int) bytes`

indices	0	1	2	3	4				n-2	n-1
values	21	-56	0	-1	109	4	5	11	-3	-23
memory	4bytes	4bytes	4bytes	4bytes	4bytes	4bytes	4bytes	4bytes	4bytes	4bytes

sizeof(int)

indices	0	1	2	3	4				n-2	n-1
values	0.11	-8.6	102	-1.8	9.75	14	1.25	101	-3.1	39.2
memory	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes

sizeof(double)

Array initialization

complete

- `int A[5]={0, 1, 10, 100, 1000}`

partial

- `int A[5] = {0,1,10} //allocates int[5]`
- only indices 0,1,2 are initialized with values

implicit size

- `int A[] = {0,1,-1,2} //allocates an int[4]`

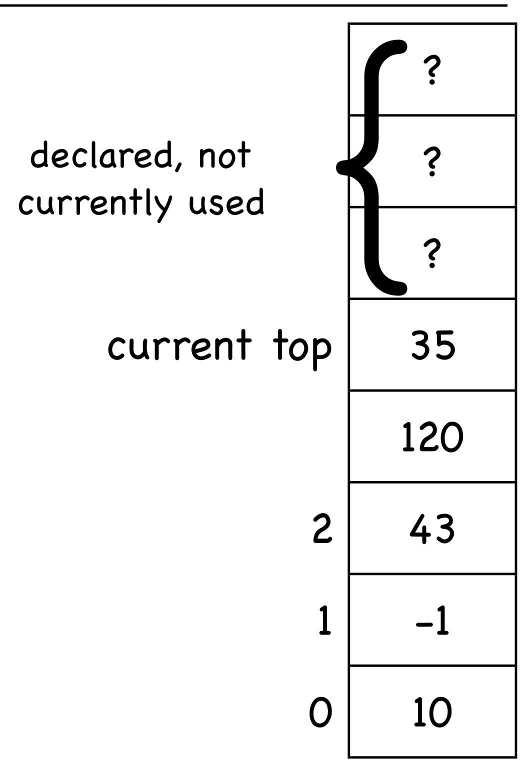
Arrays input to function

- Act as reference variable : changes made are reflected to the call array
  - in fact, it is a reference variable
  - unless defined with `const`

```
int function (int A[10], double x)

int function (int A[], double x)

int function (const int A[], double x)

int function (int A[], int array_size, double x)
```

Partially filled arrays

- similar to a stack
  - but in here we can use any element, not only the top
- int A[10] = {3, -1, 3, 41, 90}
- int current_size = 5;
- the current-top can move up or down
- example: Tower of Hanoi



Array copy

- int A[5] = {1,2,3,4,5}
- int B[5]; B=A; //error
- instead copy each element
  - `for (int i=0;i<5;i++) B[i] = A[i];`

Parallel arrays (DB tables)

ID	NAME
0	Virgil
1	Alex
2	Bob
3	Cindy
4	
n	

ID	AGE
0	34
1	22
2	18
3	31
4	
n	

ID	GENDER
0	M
1	F
2	M
3	F
4	
n	

ID	School Status
0	PhD
1	in College
2	HighSchool
3	PhD Candidate
4	
n	

- requires "join" operations

Array bounds

- <u>C++ does not check for array bounds !!</u>
- very easy to read/write at the wrong location memory address
  - writing particularly bad : can overwrite a variable

Searching
Sorting

Brute force/linear search

- Linear search: look through all values of the array until the desired value/event/condition found
- Running Time: linear in the number of elements, call it O(n)
- Advantage: in most situations, array does not have to be sorted

Binary Search

- Array must be sorted
- Search array A from index b to index e for value V
- Look for value V in the middle index m = (b+e)/2
  - That is compare V with A[m]; if equal return index m
  - If V<A[m] search the second half of the array
  - If V>A[m] search the first half of the array

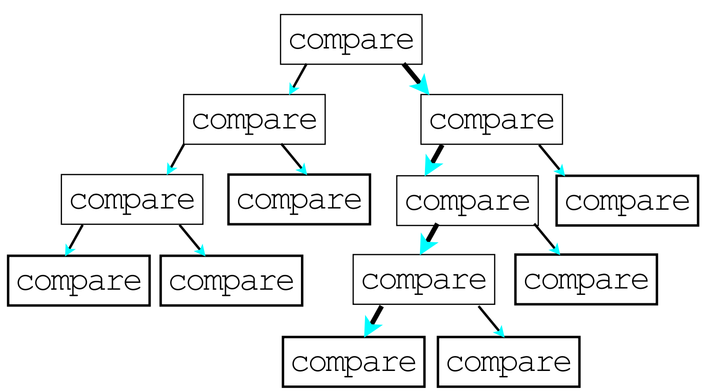
	b				m					e
V=3	-4	-1	0	0	1	1	3	19	29	47

A[m]=1 < V=3 => search moves to the right half

Binary Search Efficiency

- every iteration/recursion
  - ends the procedure if value is found
  - if not, reduces the problem size (search space) by half
- worst case : value is not found until problem size=1
  - how many reductions have been done?
  - n / 2 / 2 / 2 / . . . . / 2 = 1. How many 2-s do I need ?
  - if k 2-s, then n= 2^k, so k is about log(n)
  - worst running time is O(log n)

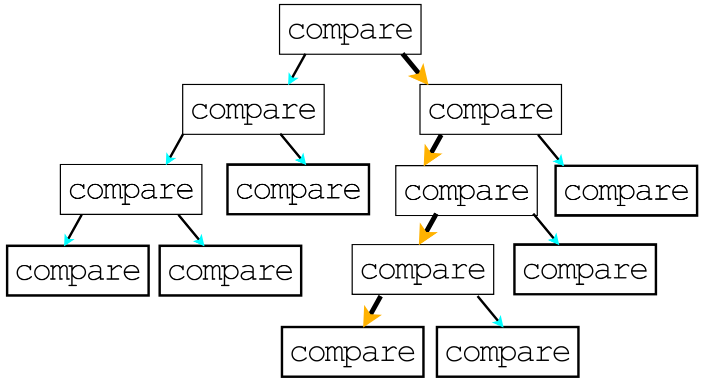
Search: tree of comparisons



- tree of comparisons : essentially what the algorithm does

•

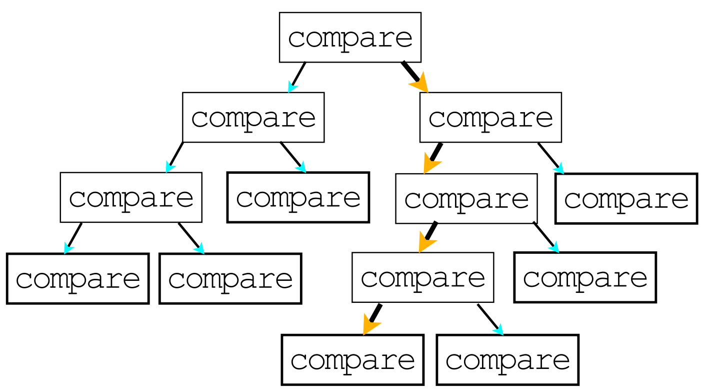
Search: tree of comparisons



tree of comparisons : essentially what the algorithm does

- each program execution follows a certain path
-

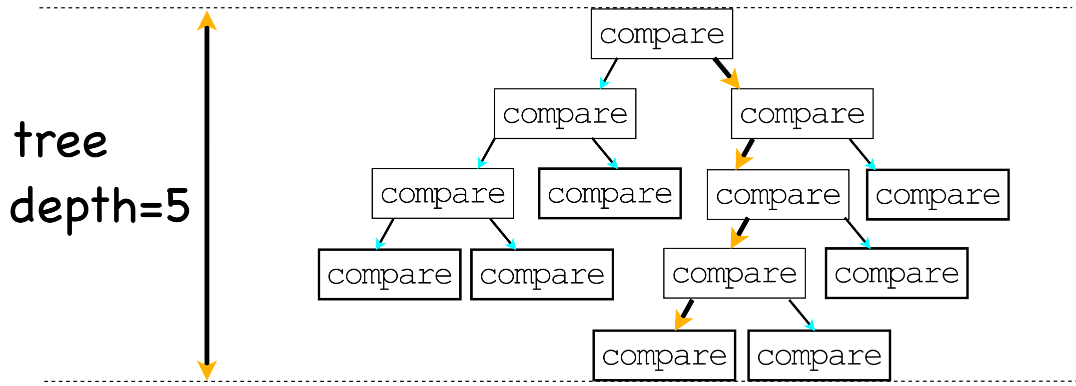
Search: tree of comparisons



tree of comparisons : essentially what the algorithm does

- each program execution follows a certain path
- red nodes are terminal / output
- the algorithm has to have at least n output nodes... why ?
-

Search: tree of comparisons



- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have n output nodes... why ?
  - if tree is balanced, longest path = tree depth = log(n)
  - if tree not balanced, path can be longer

Bubble Sort

- Simple idea: as long as there is an inversion, swap the bubble
  - inversion = a pair of indices i<j with A[i]>A[j]
  - swap A[i]<->A[j]
    - directly `swap (A[i], A[j]);`
    - code it yourself: `aux = A[i]; A[i]=A[j];A[j]=aux;`
- how long does it take?
  - worst case : how many inversions have to be swapped?
  - $O(n^2)$

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3
- move elements to make space for the new element

1	5	8		20	49				

Insertion Sort

- partial array is sorted

1	5	8	20	49					

- get a new element V=9
- find correct position with binary search i=3
- move elements to make space for the new element

1	5	8		20	49				

- insert into the existing array at correct position

1	5	8	9	20	49				

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Insertion/Selection Running Time = $O(n^2)$

used	A	C
	10	
	-1	
	-5	
	12	
	-1	
	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
	10	-5
	-1	
✗	-5	
	12	
	-1	
	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
	10	-5
✘	-1	-1
✘	-5	
	12	
	-1	
	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
	10	-5
✘	-1	-1
✘	-5	-1
	12	
✘	-1	
	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
	10	-5
✗	-1	-1
✗	-5	-1
	12	9
✗	-1	
✗	$\boxed{9}$	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
✗	$\boxed{10}$	-5
✗	-1	-1
✗	-5	-1
	12	9
✗	-1	10
✗	9	

Selection Sort

- sort array A[] into a new array C[]
- while (condition)
  - find minimum element x in A at index i, ignore "used" elements
  - write x in next available position in C
  - mark index i in A as "used" so it doesn't get picked up again
- Running Time = $O(n^2)$

used	A	C
✘	10	-5
✘	-1	-1
✘	-5	-1
✘	12	9
✘	-1	10
✘	9	12

QuickSort - pseudocode

- QuickSort(A,b,e) *//array A , sort between indices b and e*
  - q = Partition(A,b,e) *//returns pivot q, b<=q<=e*
  - *//Partition also rearranges A so that if* `i<q then A[i]<=A[q]`
  - *//* *and if* `i>q then A[i]>=A[q]`
  - if(b<q-1) QuickSort(A,b,q-1)
  - if(q+1<e) QuickSort(A,q+1,e)
- After Partition the pivot index contains the right value:

b=0 q=3 e=9

b=0			q=3						e=9
-3	0	5	7	18	8	7	29	21	10

QuickSort Partition

TASK: rearrange A and find pivot q, such that
- all elements before q are smaller than A[q]
- all elements after q are bigger than A[q]

```
Partition (A, b, e)
• x=A[e]//pivot value
• i=b-1
  for j=b TO e-1
     if A[j]<=x then
         i++; swap A[i]<->A[j]
• swap A[i+1]<->A[e]
• q=i+1; return q
```

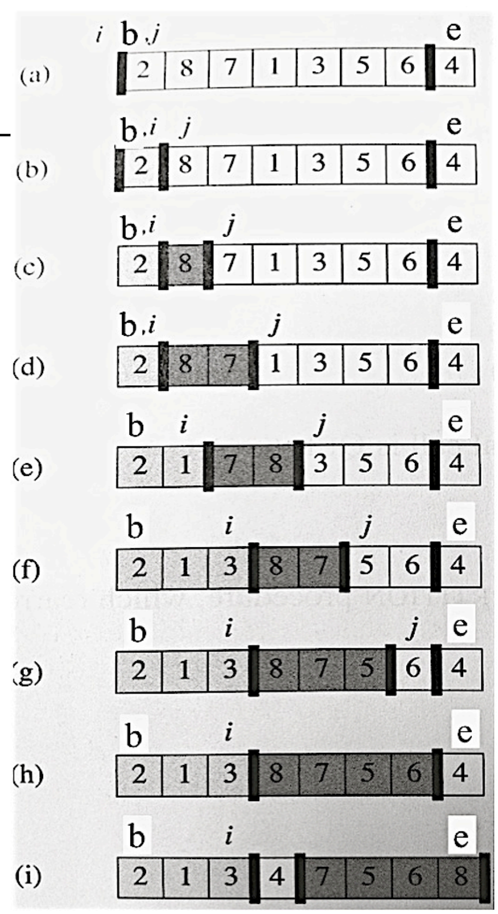
Partition Example

set pivot value x = A[e], `// x=4`
- i =index of last value < x
- i+1 = index of first value > x

run j through array indices b to e-1
- if A[j] <= x `//see steps (d),(e)`
  - swap (A[j] , A[i+1]);
  - i++; `//advance i`

move pivot in the right place
- swap (pivot=A[e] , A[i+1])

return pivot index
- return i+1



QuickSort time

- Depends on the Partition balance
- Worst case: Partition produces unbalanced split n = (1, n-1) most of the time
  - results in $O(n^2)$ running time
- Average case: most of the time split balance is not worse than n = (cn, (1-c)n) for a fixed c
  - for example c=0.99 means balance not worse than (1/100*n, 99/100*n)
  - results in O(n*log(n)) running time

Merge two sorted arrays

- two sorted arrays
  - A[] = { 1, 5, 10, 100, 200, 300}; B[] = {2, 5, 6, 10};
- merge them into a new array C
  - index i for array A[], j for B[], k for C[]
  - init i=j=k=0;
  - while (what_condition_?)
    - if (A[i] <= B[j]) { C[k]=A[i], i++ } //advance i in A
    - else {C[k]=B[j], j++} // advance j in B
    - advance k
  - end_while

MergeSort

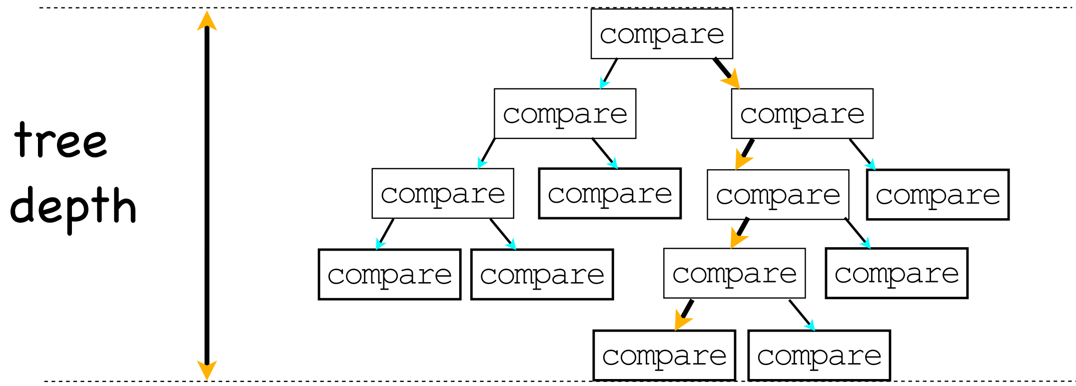
- divide and conquer strategy
- MergeSort array A
  - divide array A into two halves A-left, A-right
  - MergeSort A-left (recursive call)
  - MergeSort A-right (recursive call)
  - Merge (A-left, A-right) into a fully sorted array
- running time : O(n*log(n))

Sorting : stable; in place

- stable: preserve relative order of elements with same value
- in place: dont use significant additional space (arrays)

	time	in-place	stable
Bubble	n^2	✔	✔
Insertion	n^2	✔	✔
Selection	n^2	✘	?
QuickSort	n*log(n)	✔	?
MergeSort	n*log(n)	✘	✔

Sorting : tree of comparisons



- tree of comparisons : essentially what the algorithm does
  - each program execution follows a certain path
  - red nodes are terminal / output
  - the algorithm has to have n! output nodes... why ?
  - if tree is balanced, longest path = tree depth = n log(n)
  - if tree not balanced, path can be longer

Linear-time Sorting

- Counting Sort (A[]) : count values, NO comparisons
- STEP 1 : build array C that counts A values
  - init C[]=0 ; run index i through A
    - value = A[i]
    - C[value] ++; //counts each value occurrence
- STEP 2: build array D of positions
  - init total =0; run index i through C
    - D[i] = total;
    - total += C[i];
- STEP3: assign values to output array E
  - run index i through A
    - value = A[i];
    - position = D[i];
    - E[position] = value;

Two Dim Arrays, Vectors, Basic Hashing

Two dimensional array = matrix

- double M[10][20]; //matrix of real numbers
- allocates 10*20*sizeof(double) = 1600 bytes
- as function parameter: must specify the # of columns
  - `int myfunction (double X[][20], int rows)`
- `double M[2][5]; //allocates 80 bytes:`

indices	0 0	0 1	0 2	0 3	0 4	1 0	1 1	1 2	1 3	1 4
values	0.11	-8.6	102	-1.8	9.75	14	1.25	101	-3.1	39.2
memory	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes	8bytes

sizeof(double)

Two dimensional array = matrix

- `double C[10][20],B[10][20];`
- `C = A + B; //error`
  - instead compute each element

```
for (int i=0; i<10; i++)
for (int j=0; j<20; j++)
    C[i][j] = A[i][j] + B[i][j];
```

- `double D[20][5],E[10][5];`
- `E = A*D; //error`
  - instead compute each element

```
for i (rows of A)
for j (columns of D)
    C[i][j]=0;
    for k (columns of A)
        add component A(i,k)B(k,j) to C[i][j] ;
```

Matrix determinant

- Recursive formula. Fix a row i:

$$|A| = \sum_{j=1}^{m} (-1)^{i+j} a_{ij} M_{ij}$$

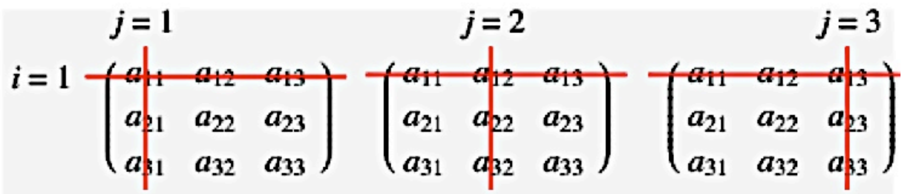
- where M_{ij} is the determinant of the matrix obtained from A by removing row i and column j

Matrix determinant

- Recursive formula. Fix a row i:

$$|A| = \sum_{j=1}^{m} (-1)^{i+j} a_{ij} M_{ij}$$

- where M_{ij} is the determinant of the matrix obtained from A by removing row i and column j



For example, for a 3 x 3 matrix, the above formula gives

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \cdot ($$

Vectors

- part of Standard Template Library (STL)
  - some compilers do not support it
  - some methods work differently on different compilers

```
#include <vector>
vector <int> a; //initial size 1 int
vector <int> b(10); //initial size 10 ints
vector <int> c(10,1); //initial size 10 ints all initialized with value 1
vector <int> d(b); //initial size 10 ints, having the same content as vector b
```

- vectors change size dynamically: they automatically allocate more memory when need it
  - `d[14]=1452;` //this is still out of bounds

Vectors

- array syntax works
  - more options available
- `vector <double> x(20,0);`
- `x[2]=-90.67; x[4]=1.46; x[6]=0.66`
- `for (int i=0;i<x.size();i++)`
  - `cout<<" x["<<i<<"]="<<x[i];`

Vector methods

.at(index), [index]	returns the element
.push_back(value)	adds a value as last element
.pop_back()	removes last element
.size()	returns the size
.clear()	removes all elements
.empty()	returns true if vector is empty, false if not
.reverse()	reverse the order of elements
.resize(extra, value)	adds extra new elements, all init with value
.swap()	swap content of two vectors

Vectors VS Arrays

- vectors are passed by value, while arrays are passed as references parameters to functions
  - vectors can be also passed as reference using "&"
- with arrays its easy to read/write wrong memory address
  - vectors have some protection mechanism
- vectors dynamically allocate more memory when need it
- vectors are a `class`,so they have methods
- whenever possible, use vectors for software development

Searching and sorting Vectors

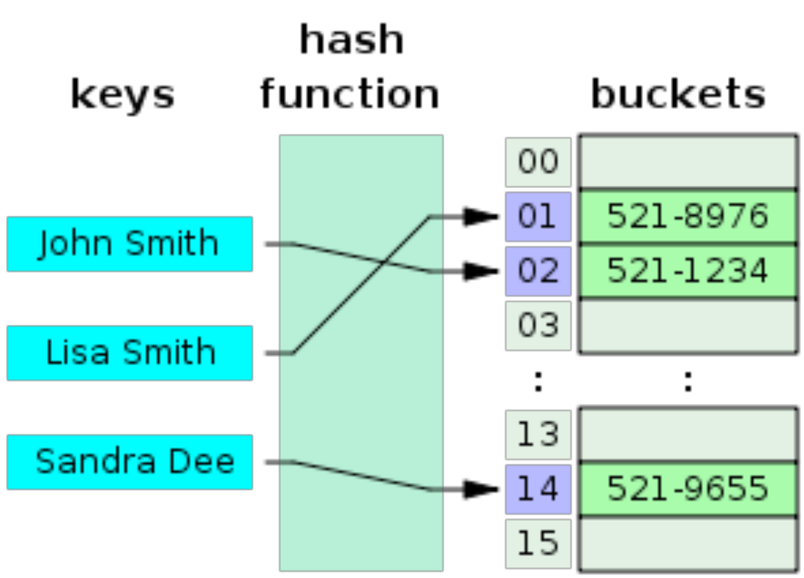
- in principle, same algorithms like before
  - and more: max, min, median
- implementation: by default vectors passed by value
  - therefore function-changes to argument does not reflect back to the call vector
  - solution: pass by reference
  - solution: use global variables
  - solution: use return values

Basic hashing

- arrays are very nice, but keys have to be integers
  - keys from 0 to N-1
- hash function: take input any key, returns an index (int)
- very useful when natural keys are not integers
  - names, words, addresses, phone numbers etc
  - even if key=integer (like phone #) they are not the integers we want as indices
- text processing : natural keys are words/n-grams/phrases
- databases: natural keys can be anything

Hash Tables

- key -> index -> lookup in array / table



Hash function: two qualities

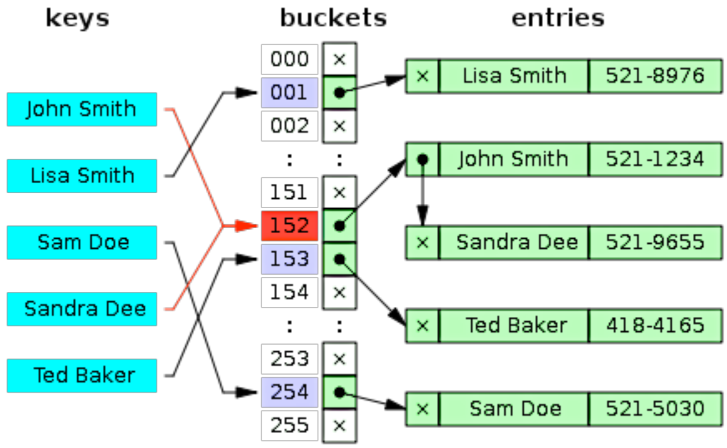
- int hash_function (char[])
- quality ONE: one-to-one (injection). Different inputs result in different outputs
  - collision: having many words map to same index
  - collisions eventually will happen, need to be solved
  - collisions should be balanced (uniformly distributed) per output indices
- quality TWO: the set of returned indices must be manageable
  - for example returns integers from 1 to 100000
  - or returns integers in range (0, MAXHASH)

Simple hash function

- return a simple combination of characters, modulo MAXHASH
- int MAXHASH=100000;
- `int hash_function(char[])` // returns integers between 0 and MAXHASH
  - `int sum=0,i=0;`
  - `while(char[i]>0) {sum+=char[i] * ++i*i;}`
  - `return sum % MAXHASH;`

Hash Tables - Collisions

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key



Hash Tables - Collisions

- when several keys (words) map to the same key (index)
- have to store the actual keys in a list
  - list head stored at the index
- key -> index -> list_head -> search for that key