CST8130 – Data Structures

Professor : Dr. Anu Thomas
Email: thomasa@algonquincollege.com
Office: T314

Hashing

Linked Lists vs Arrays

Storing data in a linked lists was an **improvement** over arrays with memory:

- we did not have a solid block of consecutive memory addresses to hold the elements (or references to the elements);
- we were not constrained by size (arrays needed us to declare a fixed size – which we could then accommodate by reallocating larger array and copying existing array to new larger one – overhead)
- But…we had to live with extra references per element for next/previous

Algorithm efficiency with Linked List was O(1) if we could add/delete/search at either head or tail

But with algorithm efficiency: no improvements if we need "sorting"

	Array – best case:	Linked List – best case
insertion in order	O(*n*)	O(*n*)
searching	O(*log n)*	O(*n*)
deletion	O(*n*)	O(*n)*

Linked List - problems

Storing data in a linked lists was NOT an improvement over arrays in that

- we lost our ability to access any element directly – we had to “follow the links” to find an element – hence we lost our ability to perform binary search

Trees gave back the ability to “binary search”

Binary Search Trees

Search – O(log *n*) – which is quite suitable for large data sets (if the tree is balanced)

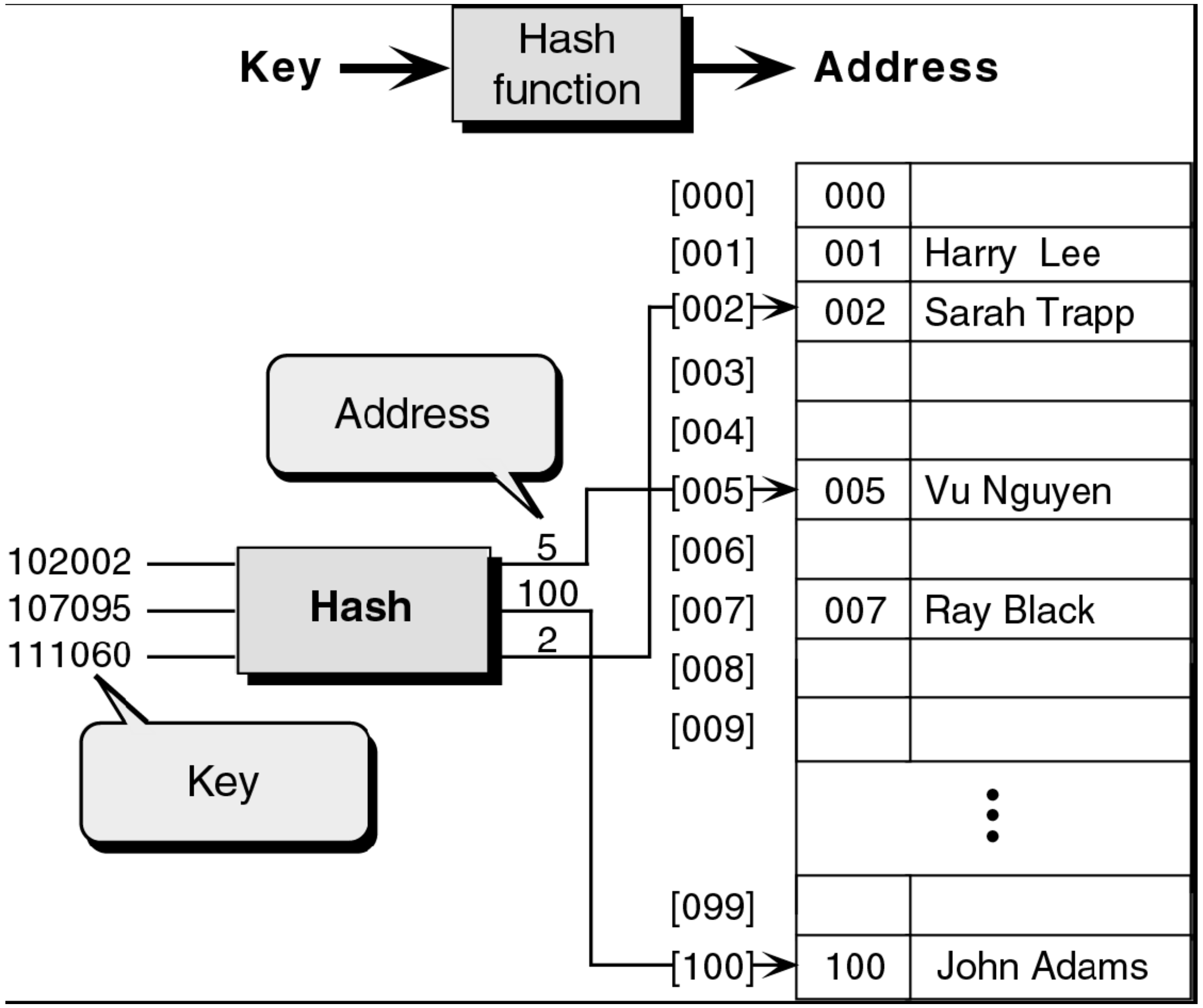
Insertions - O(log *n*) – which is quite suitable for large data sets (if the tree is balanced)

Unbalanced trees approach O(*n*) efficiency for searches and insertions

Plus, there is still the overhead of two references with every node.

Can we do better?

Overview of Hash Function/Searching



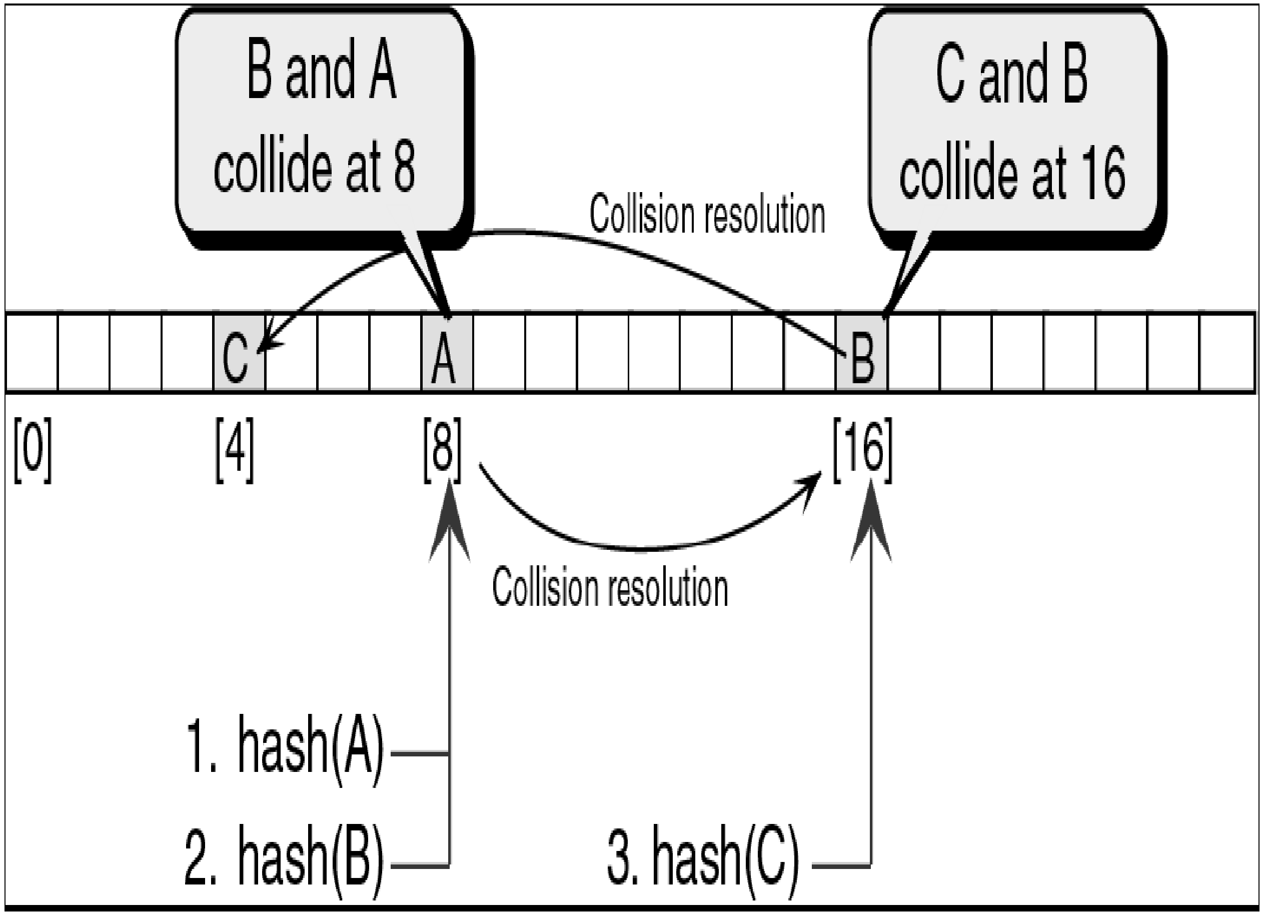
Array organization

Access array elements with index

Convert "key" into index

Insertion/Search - O(*1*)

Potential Problem - Collisions



Collisions occur when multiple keys generate same index

Need mechanism to resolve

Hashing Methods

Direct – key is address without any algorithmic manipulation

con – there is an element for every possible key value

pro – no collisions

Subtraction – when keys are consecutive but do not start at 1

– subtract a base number from key to give direct hashing.

Example all student numbers start with 040 --- ---

Modulo Division – divides the key by the array size, and uses the remainder as the index.

Digit Extraction – selected digits are extracted from key and used as index

Midsquare – key is squared and the address is selected from the middle of the squared number. Problem is result of squaring can overflow your data type

Hashing Methods (contd.)

Folding – fold shift – key value is divided into parts whose size matches the index size. Then the left and right parts are shifted and added with the middle part.

Example: "123456789" is divided into "123", "456", "789" . These are added together to get 1368.

- **fold boundary** – key value is divided into parts; the left and right parts are "reversed" and added to middle part.

Example: "123456789" is divided into "123", "456", "789" . The left and right are reversed to "321" and "987" and added to "456" to get 1764.

Rotation – usually used in combination with other methods – move part of end of key to front to get index

Pseudorandom generation – key is used as seed in random number generator, resulting random number is scaled into possible index range using modulus division.

Aim of Hash Functions

Want to find an algorithm (can be composed of a sequence of multiple methods) that will map the key values as uniquely as possible to index values (ie minimize collisions)

Requires knowledge of key values and their distribution.

Often, adjust the size of the array to get best results.

Many key values are strings of chars – but they can be converted to numeric by using arithmetic operations on each char’s ASCII value

Load factor – number of elements in list divided by number of physical elements stored in list (as a percentage)

Collision Resolution

Open Addressing – resolves collisions in the prime area, as opposed to using overflow area

a) **Linear Probe** – add one to current index and try again

b) **Quadratic Probe** – increment to hash index is the collision number squared. So, for first collision – add 1^2 to index; for second collision – add 2^2 to index; for third collision, add 3^2 to index, etc. Note – we may need to use modulus division in case computed index becomes too big

c) **Pseudorandom Resolution** – uses the collision index (instead of the initial key) as the seed in a random number generator

d) **Key-Offset** – calculates new index as a function of the old index and the original key.

Linked List – each element in the prime area contains a pointer to a linked list of elements. When a collision occurs, if is inserted into the linked list for that hash index. For efficiency, we need to watch size of the overflow area (ie how many, and size of linked lists)

Buckets – Each index location can hold several records – effectively increases size of prime area

Questions?