

DATA STRUCTURES, Dynamic Sets ADT

- **Goal:** investigate **data structures and algorithms** that support efficient implementation of various operations on sets.
- **Dynamic sets:** may change over time.
- **Key** identifier of an element.
- **Dynamic set ADT** , a collection of elements and operations that work on them
- **General operations on a dynamic set ADT:**
 - *Search*
 - *Insert*
 - *Delete*
 - *Minimum*
 - *Maximum*
 - *Successor*
 - *Predecessor*

Dictionary ADT

■ Go look it up!

■ Primary use : store elements so they can be located quickly using *keys*

The element may store other useful information *data*.

Examples:

- A dictionary of bank accounts, key is account number
- A dictionary of a set of windows opened in a graphical interface, key is a window ID
- A dictionary of student database, key is student ID
- *Symbol table* used by compilers

■ Dictionary ADT: A dynamic set which supports, *Search, Insert, Delete*.

■ Hash Table: a data structure that is used to implement Dictionary ADT.

- typically, worst-case time to perform the operations is $O(n)$
- expected time is $O(1)$

Dictionary ADT

- **Dictionary of records** Records have distinct keys,
 x is a reference to a dictionary record

x key

data

- **Operations** we will consider:

- $insert(T, x)$, insert the record pointed to by x into the dictionary
- $delete(T, x)$, remove record pointed to by x from the dictionary
- $search(T, key)$, return pointer to record with given key, if it is in the dict., or return **null** if no record has that key.
- $update(T, oldrec, newrec)$, update the record pointed to by $oldrec$ to have the data of the record pointed to by $newrec$

$x = search(key)$

$delete(x)$

Implementing Dictionary ADT

- Some ideas...

Let n be the number of elements in the dictionary

- Time complexity

	insert	search	delete	update	space
unsorted dbl-linked	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
sorted dbl-linked	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted arr	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

- Can we do better? – “Go look it up!”, need faster search.

Implementing Dictionary ADT (*continued*)**■ Direct-address table** (DAT)

- $key \in \{0, 1, \dots, r - 1\}$
- Example: $r = 5$

DAT	
0	
1	1
2	2
3	
4	4

- *Element with key k is stored in slot k .*
If $T[i] = NULL$, no record with key i .

Implementing Dictionary ADT (*continued*)

- Time Complexity for DAT

T , direct address table,

```
DirectAddress_Search( T, key k)
    return T[k]
```

```
DirectAddress_Insert( T, ptr x to element)
    T[key(x)] = x
```

```
DirectAddress_Delete( T, ptr x to element)
    T[key(x)] = NULL
```

Each operation $\Theta(1)$ time.

- Moderate r , $r < 1000$. What is r is too big?
 What if the number of keys, n , stored at any particular time \ll total possible number of keys, r ?
 Example: student dictionary, $r = 10^9$, $n = 4000$

■ **Hash Table** A variation of the idea of DAT, where the element with key k is stored at slot $h(k)$ instead at k .

The keys do not have to be integers.

Hashing

- **Hash Tables** (HT) are typically one of the most efficient ways of implementing a Dictionary ADT, particularly if we know something about the distribution of the key values.

HT do not support efficiently operations that rely on relative order of data elements.

- The keys are from an *universe* U .

- **Hash function** , maps keys to integers,

$$h : U \rightarrow 0, 1, \dots, m - 1, \text{ typically } |U| \gg m$$

Hashing (<i>continued</i>)

- **Collision** , two keys hash to the same slot.
Cannot avoid it.

- **Collision resolution by chaining**

Put all elements that hash to the same slot in an unsorted doubly-linked list, where the hash table entry is a ptr to the first item in the list

$T(h(k))$ - a ptr to the head of the list containing the element with key k .

If $T(h(k)) = NULL$, there are no elements with key k in the dictionary.

Hashing (<i>continued</i>)

- Some possible implementations of the operations:

```
ChainedHash_Search( T, key k) // T is a hash table  
    search for element with key k in the list T[h(k)]
```

```
ChainedHash_Insert( T, ptr x to element)  
    insert x at the head of the list T[h(key(x))]
```

```
ChainedHash_Delete( T, ptr x to element)  
    delete x from the list T[h(key(x))]
```