Data Structures and Algorithms – W06

Linear and binary search

Priority queue

Hashtable

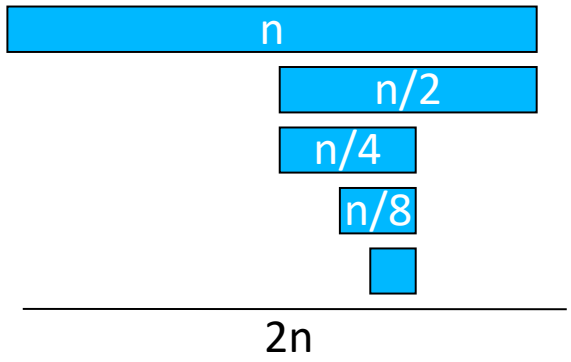
Contents

- Searching for the median and i-th element
- Linear search
- Binary search
- Priority queues:
  - Using linked-list
  - Using arrays
  - Using heap
- Hashing tables
  - Open addressing
  - Array of lists

Searching for the median and i-th element

- The minimum / maximum search is a part of the selectSort algorithm with the complexity O(*n*)
- How to find the median (middle value) or i-th element of the sorted sequence. First sorting? Therefore, the complexity: O(*n*log*n*)
- This can be done faster using the idea of quicksort.

```
// A – array, p, r  – scope of search
// i  - search position
RANDOMIZED-SELECT(A, p, r, i)
  if p = r
    return A[p]
  q=RANDOMIZED-PARTITION(A, p, r)
  k=q - p + 1
  if i = k  // the pivot value is the answer
    return A[q]
  else if i < k
     return RANDOMIZED-SELECT(A, p, q - 1, i)
 else
    return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```



Average complexity: O(*n*)

Linear search(LS)

- In a linear unordered collection to search for an item with a given content, check each item moving from the beginning to the end of the collection. This is a **linear search**.
- Worse-case complexity, average complexity: O(n)
- In order to compare the elements with the searched pattern, you can use:
  - the `equals()` method - but there can only be one such method
  - comparator

```
public class Searching<T> {
/** returns the position of the found element if there are no such an
element returns -1*/
  public int linearSearch(IList<T> list, Comparator<T> comp, T what ){
    int pos=0;
    for(T elem:list){
      if(comp.compare(what,elem)==0)
        return pos;
    else
      pos++;
    }
    return -1;
  }
}
```

Binary search (BS)

- If we have a linear sorted collection with random access (according to the value we are looking for), we can use a **binary search**.
- We compare the searched value with the middle element. If the search value is smaller - it is located to the left of the middle position. If bigger - to the right. If equals - the end of the search. With the part where the value can be searched, we proceed exactly the same as with the initial collection. If part of the collection becomes empty - there are no values in the collection at all.
- Worse-case ad average complexity: O(log *n*)

```
public int binarySearch(IList<T> list, Comparator<T> comp, T what ){
  int left=0;
  int right=list.size()-1;
  int middle;
  while(left<=right){
    middle=(left+right)/2;
    int compValue=comp.compare(what,list.get(middle));
    if(compValue==0)
      return middle;
    if(compValue<0)
      right=middle-1;
    else
      left=middle+1;
  }
  return -1;
}
```

Sorted linked-list

- Linear structure without random access and binary search:
  - The implementation shown will result in the complexity of O (*n*log *n*) due to `get` operations
  - Using list iterators, you can move forward and backward instead of using the `get` operation, which will limit the complexity of moving to O(*n*), although there may be up to twice as many steps as in a linear search.
  - The number of comparisons will be limited to O(log *n*).
  - Conclusion: the correct binary search algorithm for lists makes sense when the comparison time is definitely greater than the time of moving through the list.

Priority Queue (PQ)

- Data structure in which we need two basic operations:
  - Add an item to the collection
  - Get and delete the item with the highest priority.
- Additionally, the operation may be expected:
  - Get the largest element of the collection (without removing)
- Because the operation names are the same as for the FIFO queue (although they have different meanings), you can use the `IQueue` interface as an interface for priority queues.

Priority queue – linked list

- Implementation of the priority queue using a linked list:
  - unordered:
    - Inserting an element (at the end or to the beginning): O(1)
    - Get and remove an item - first you need to find an item using a linear search and then "unlink" from the list: O(*n*) + O (1) = O (*n*)
  - Ordered by priority:
    - Insert element (based on priority): O(*n*)
    - Get and remove an item from the beginning of the list: O(1)

PQ – linked-list – code 1/2

```
public class UnsortedListPriorityQueue<T> implements IQueue<T> {
  private final TwoWayCycledListWithSentinel<T> _list;
  private final Comparator<T> _comparator;

  public UnsortedListPriorityQueue(Comparator<T> comp) {
    _comparator=comp;
    _list=new TwoWayCycledListWithSentinel<T>();
  }

  public boolean isEmpty() {
    return _list.isEmpty();
  }

  public boolean isFull() {
    return false;
  }
  public int size() {
    return _list.size();
  }
  public T first() throws EmptyQueueException {
    if(_list.isEmpty())
      throw new EmptyQueueException();
    return _list.get(getIndexOfLargestElement());
  }
```

PQ – linked-list – code 2/2

```
  public void enqueue(T elem) throws FullQueueException {
    _list.add(elem);
  }
  private int getIndexOfLargestElement(){
    if(_list.isEmpty())
      return -1;
    Iterator<T> iter=_list.iterator();
    int counter=0, maxPos=0;
    T value=iter.next();
    T elem=null;
    while(iter.hasNext()){
      counter++;
      if(_comparator.compare(elem=iter.next(), value)>0){
        maxPos=counter;
        value=elem;
      }
    }
    return maxPos;
  }
  public T dequeue() throws EmptyQueueException {
    if(_list.isEmpty())
      throw new EmptyQueueException();
    return _list.remove(getIndexOfLargestElement());
  }
}
```

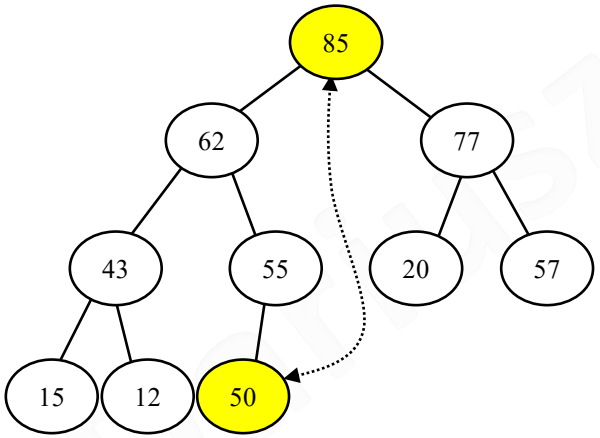
Priority queue – an array

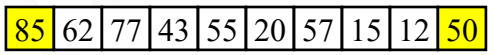
- Implementation of the priority queue using the table:
  - Unordered:
    - Insert an element (last position), amortized complexity: O(1)
    - Get and remove an element - first you need to find an element using a line search and then move the elements to the left in the table: O(n)
  - Ordered by priority (in reverse order):
    - Insert an element - finding an item using a binary search, then moving elements to the left in the table: O(log *n*) + O(*n*) = O(*n*)
    - Get and remove the item - it is at the end, therefore O(1)

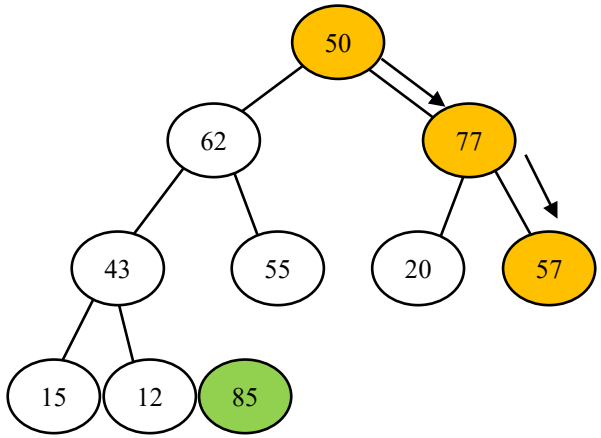
Priority queue – a heap

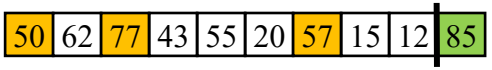
- Heap – a structure from the previous lecture - can be used to support the operation of the priority queue:
  - The operation of picking the element is part of one step of the sorting algorithm - when we have the correct heap we should replace the root with the last element of the array and repair the heap: O(log *n*)
  - Repair of the heap is the "sinking" of the element

```
swap(heap,n-1,0);
sink(heap,0,n-1);
```



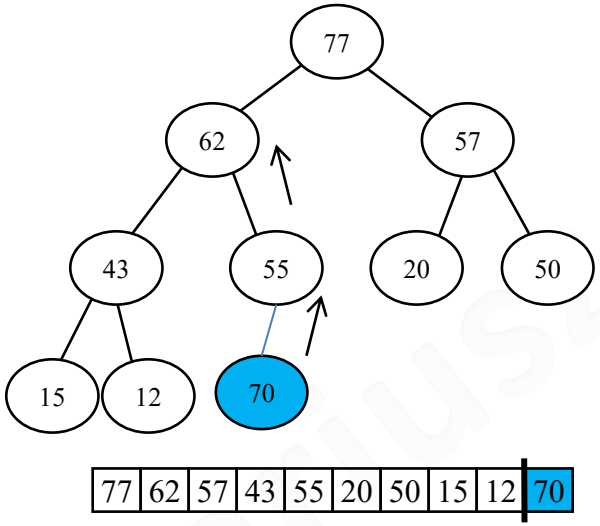


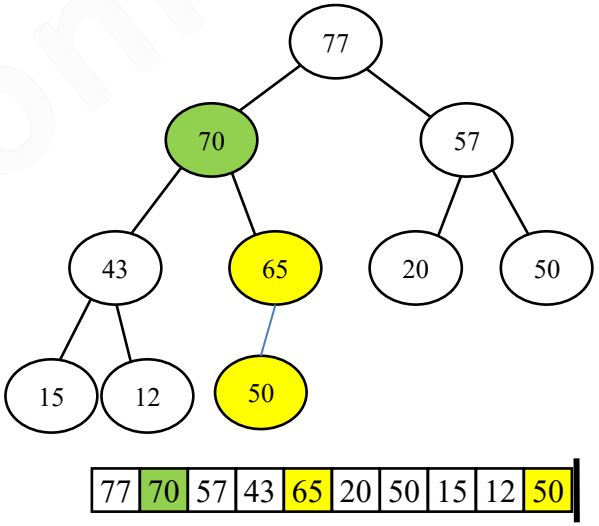




Heap – insert element

- Inserting a new item to the heap:
  - We add on the next free position in the table
  - We treat it as a new leaf, which, however, may break the principle of the correct heap.
  - We repair the heap going from the inserted leaf upwards (swim).
- Complexity: O(log *n*)





Heap as PQ – code 1/2

```
public class HeapPriorityQueue<T> implements IQueue<T> {
private final IList<T> _list;
private final Comparator<T> _comparator;
public HeapPriorityQueue(Comparator<T> comparator) {
  _comparator = comparator;
  _list = new ArrayList<T>();}
public void enqueue(T value) {
  _list.add(value);
  swim(_list.size() - 1);}
public void clear(){
  _list.clear(); }
public int size(){
  return _list.size(); }
public boolean isEmpty(){
  return _list.isEmpty();}
public boolean isFull() {
  return false;}
public T first() throws EmptyQueueException {
  if (isEmpty()) throw new EmptyQueueException();
  return _list.get(0);}
public T dequeue() throws EmptyQueueException {
  if (isEmpty()) throw new EmptyQueueException();
  T result = _list.get(0);
  if (_list.size() > 1) {
    _list.set(0, _list.get(_list.size() - 1));
     sink(0);}
  _list.remove(_list.size() - 1);
  return result;}
```

Heap as PQ – code 2/2

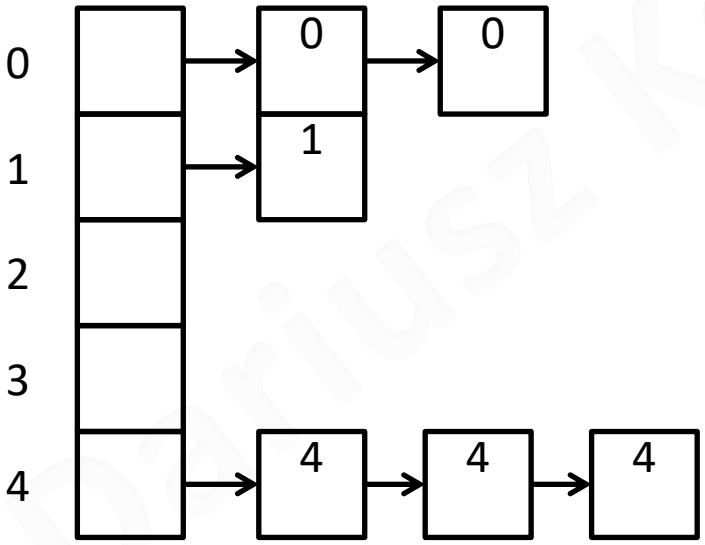
```
private void swap(int index1, int index2) {
  T temp = _list.get(index1);
  _list.set(index1, _list.get(index2));
  _list.set(index2, temp);
}

//moving up the element, iterative version
private void swim(int index) {
  int parent;
  while(index != 0 &&
    _comparator.compare(_list.get(index), _list.get(parent= (index - 1) / 2)) > 0){
       swap(index, parent);
       index=parent;
  }
}

// sinking the element down the stack, iterative version
private void sink(int index) {
  boolean isDone=false;
  int child;
  while(!isDone && (child=2*index+ 1 ) < _list.size()) {
    if (child < _list.size()- 1 &&
      _comparator.compare(_list.get(child), _list.get(child+1)) < 0)
        ++child;
    if (_comparator.compare(_list.get(index), _list.get(child)) < 0){
      swap(index, child);
      index=child;
    }
    else isDone=true;
  }
}
```

PQ – other implementations

- If there are few priorities, let's assume *k* values, the easiest way is to implement PQ as an array (or list) of lists **for each priority separately**. For simplicity, let's assume that these are integers ranging from 0 to *k*-1
  - Inserting an element for such a list array is O(1)
  - Getting the item - in the worst case you need to check one by one which list is not empty - O(*k*)=O(1)



Complexity comparision realizations of PQ

Implementation	enqueue()	dequeue()	first()
Unordered array	O(1)	O(n)	O(n)
Ordered array	O(n)	O(1)	O(1)
Unordered linked list	O(1)	O(n)	O(n)
Ordered linked list	O(n)	O(1)	O(1)
Heap on array	O(log n)	O(log n)	O(1)
Array of lists	O(1)	O(k)	O(k)

Dictionary

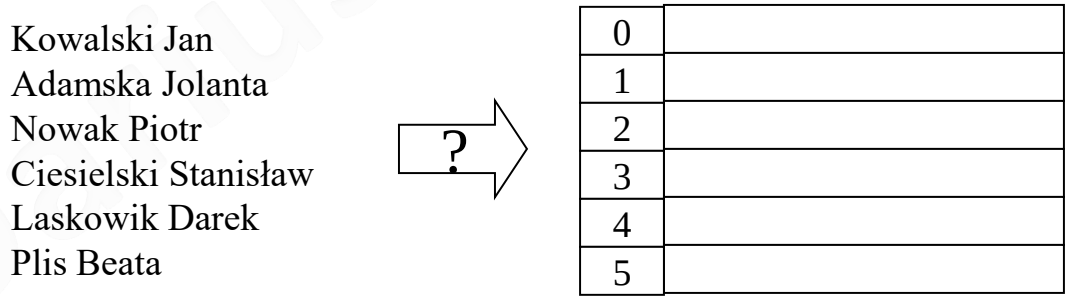
- Dictionary is a dynamic (usually) structure of data to remember:
  - keys
  - key-value pairs.
- The dictionary is also known as an associative array, an association table, an array indexed with a key, projection, and a map.
- The dictionary must allow two basic operations that must be as effective as possible:
  - Finding an element based on a key - the most often operation!
  - Addition of an element
- The key dictionary returns only logical information if the key is or is not. In the key-value dictionary: the value associated with the key (or **`null`**).
- A static dictionary has only one operation:
  - Finding an item by key.
- The dictionary may also allow other operations (or a subset of them):
  - Removing an element by key
  - Review of the dictionary in a way sorted by key
  - Combining dictionaries into a new dictionary

Dictionary using tables/lists

- Unordered list:
  - Finding element - linear search O(*n*)
  - Adding element - at the end O(1)
- List ordered on the linked list:
  - Finding the element: linear search O(*n*), on average half of the elements will be compared
  - Adding element: linear search O(*n*)
- List ordered on an array:
  - Finding the element: binary search O(log *n*).
  - Adding element: O(*n*) due to shifting elements in the array.

HashTable (HT)

- Is it possible to find the key faster than in O(log *n*) time? Yes!
- A hash table!
- Idea:
  - An array of fixed length can contain an element or value "empty" (eg, **`null`**) under a particular index.
  - The array has a hash function that returns a integer value for each key.
  - Inserting an element: the value of the key position (key-value pair) is determined by the value of the hash function modulo size of the array.
  - Finding an element: a key is an argument to a hash function that returns a position in an array. We check whether the searched key is saved in the given position:
    - If YES - the key is in the set (or we return the value from the key-value pair)
    - If NO - the key is not in the set (or we return the value "empty")

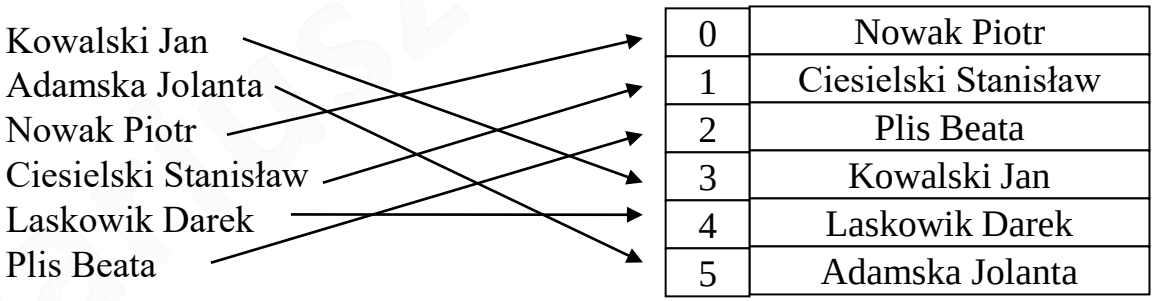


HT - example

K – key (here name)
h(K) = K[0] – generate an index using first letter of a name (ASCII code)
pos(K) = h(K) mod 6 - position obtained from has function modulo size

h(„Kowalski”) = („K”) mod 6 = 75 mod 6 = 3
h(„Adamska”) = („A”) mod 6 = 65 mod 6 = 5
h(„Nowak”) = („N”) mod 6 = 78 mod 6 = 0
h(„Ciesielski”) = („C”) mod 6 = 67 mod 6 = 1
h(„Laskowik”) = („L”) mod 6 = 76 mod 6 = 4
h(„Plis”) = („P”) mod 6 = 80 mod 6 = 2

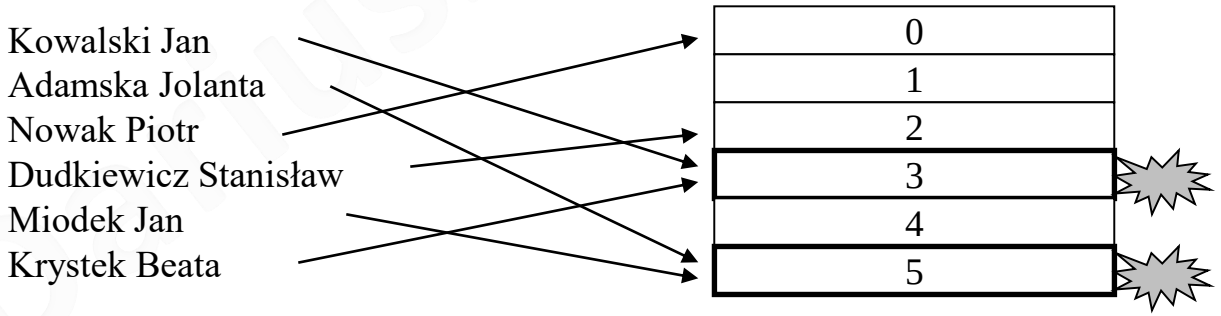
A - 65	G - 71	M - 77
B - 66	H - 72	N - 78
C - 67	I - 73	O - 79
D - 68	J - 74	P - 80
E - 69	K - 75	Q - 81
F - 70	L - 76	R - 82



HT – not always ideal

- A perfect hash function is a function that always generates different array indices for different keys
- In principle, only possible for a fixed set of keys (static hash table):
  - Keywords of a given programming language
  - A set of observed words in the stream
  - e.t.c.
- Special techniques for searching for such functions

- When a set of keys is dynamic (eg increases) or finding a perfect function conflicts are difficult:



HT – open addressing

- Collision resolution in open addressing:
  - In addition to the hash function h(*K*), there is an incremental function to indicate the next position to be checked in the *i*-th and attempt: p(*i*)
  - In the event of a collision, the following positions are checked successively (h(*K*) + p(1))% size, (h(*K*) + p(2))% size, etc.
  - Analogous positions must also be checked when searching for an item

HT – linear probing

- p(i)=i
- it means, in i-th probe we check ($h(K)+i$) mod *TSize slot*

A_5 – a key, for which the value of hash function is 5

A_5 A_2 A_3

0	
1	
2	A_2
3	A_3
4	
5	A_5
6	
7	
8	
9	

B_5 A_9 B_2

0	
1	
2	A_2
3	A_3
4	B_2
5	A_5
6	B_5
7	
8	
9	A_9

B_9 C_2

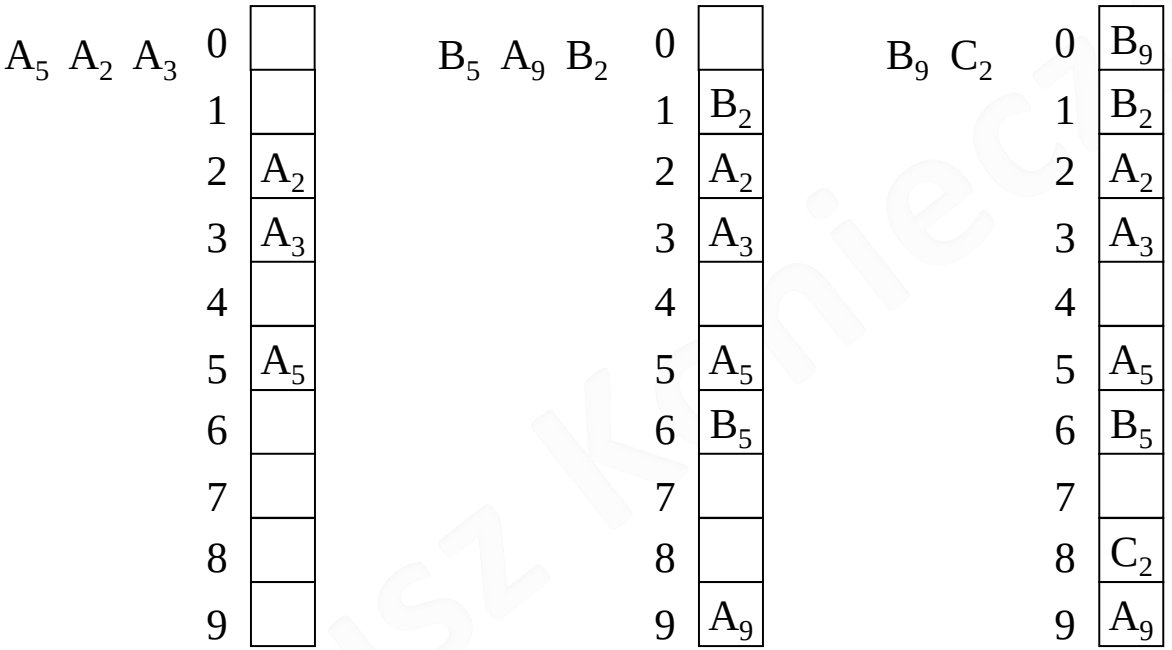
0	B_9
1	
2	A_2
3	A_3
4	B_2
5	A_5
6	B_5
7	C_2
8	
9	A_9

• Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***.

- Searching procedure for A_2,C_2
- Searching procedure for A_4
- Removing procedure for B_2

HT – Quadratic probing, double hashing

- $p(i)=c_1 i+c_2 i^2$
- or $p(i)= (-1)^{i-1}((i+1)/2)^2$ (+1, -1, +4, -4, +9, -9, ...)



Quadratic probing leads to ***secondary clustering***.

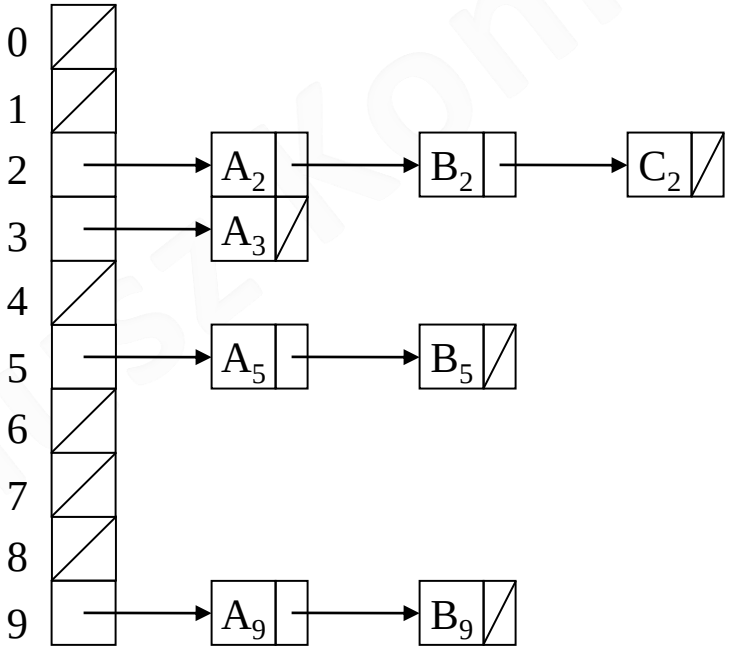
Solution to the problem of clustering elements:
Double hashing (incremental function also depends on the key):): $p(i,K)=i*h_2(K)$

Open addressing - removing

- Removing an item with the given key:
  - It may not occur at all
  - If it is rare, we can proceed as explained in this lecture
  - If it can be frequent, you can add a marker to each position of the array, if it has been removed (once in a while you do the real deletion - reorganize the table)
    - You can insert a new item in the position with a marker
    - When looking for an element, the position with the marker is treated as occupied.

HT as an array of linked lists

- Instead of using open addressing, it's better to use an array of lists (better performance at the expense of memory).
  - Each position of the array a linked-list (eg two-way circular with a sentinel)
  - Instead of conflicts we have to insert an element into the list indicated by the hash function (eg at the end in time O(1)).
  - Deleting is deleting from the linked list
  - Search is search on a linked list
  - The last two operations depend on the length of the list, but assuming a good hash function and a small degree of filling the array, these lists will be very short in the order of 2-3 elements.



Hash function

- Hash function - the first important component of hash tables:
  - For the same key must always return the same value
  - For similar keys should give very different values
  - It should be quick to calculate

Load factor – α

$$\textbf{Load factor } \alpha = \frac{\text{number of used slots}}{\text{size of table}}$$

- Marked as α - this is the number of elements in the array to all table items. Hence, it is always in the range [0, 1].
- The smaller the value of α, the lower the chance of collision.

HT – Complexity O(f(α))

- The complexity depends on the load factor (of course for large tables) and the presence (or not) of the element in the dictionary

Algorithm	Attempts for el $\in$ dictionary	Attempts for el $\notin$ dictionary
Array of lists	$1+\alpha/2$	$1+\alpha$
Linear probing	$0,5+1/(2*(1-\alpha))$	$0,5+1/(2*(1-\alpha)^2)$
Double hashing	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$

HT – Complexity - examples

Algorithm-situation	Load factor - α			
	1/2	2/3	3/4	9/10
Array of lists - success	1,25	1,33	1,38	1,45
Array of lists - defeat	1,5	1,67	1,75	1,9
Linear probing- success	1,5	2,0	3,0	5,5
Linear probing- defeat	2,5	5,0	8,5	55,5
Double hashing - success	1,4	1,6	1,8	2,6
Double hashing - defeat	2,0	3,0	4	9

HT - summary

- Hash tables are the best as dictionaries.
- However, they do not provide any order.
- Fixed hash tables are used, for example, in compilers.
- For correct complexity, a good hash function must be selected

- Technical information:
- Java already in the `Object` class has a `hashCode()` method to generate values of hashfunction:
  - This method normally depends on the reference, which is not what we expect. If you want to use hash tables, you should overwrite it with your own version.
- Many classes have the correct version of this method, e.g. `String`. In creating your own hash method, it's a good idea to use values from the `hashCode()` method of your own class fields, combining them with an XOR bitwise operation. For example, this has been implemented in the `AbstractList<T>` class:

```
// ^ - exclusive or
  @Override
  public int hashCode() {
    int hashCode = 0;
    for (E item:this)
      hashCode ^= item.hashCode();
    return hashCode;
  }
```