Linear Data Structures (Sequences)

Definitions:

- A sequence of values of type T is a mapping from an index set [istart,..., iend] into a set of values of type T
  (array)
  => static environment

- A sequence of values of type T is either empty or a pair consisting of an element of type T, followed by a sequence of values of type T
  (linked list)
  => dynamic environment

Arrays

Abstract Data Types

Stack
- **insert (push)**
- **return last inserted object (top)**
- **delete last inserted object (pop) => LIFO**
- **check if stack is empty**
- **check if stack is full**

Queue
- **insert (enqueue)**
- **return first inserted object (front)**
- **return last inserted object (back)**
- **delete first inserted object (dequeue) => FIFO**
- **check if queue is empty**
- **check if queue is full**

Set
- **insert**
- **search**
- **delete**
- **check if set is empty**
- **check if set is full**

Searching

Specification

Pre Condition:

- **Array f of mutually distinct keys with index set [istart, ..., iend]**
- **Search key s**

Post Condition:

- **information if s is contained in f**
- **index ifound of s in f**

Measures of Complexity:

- **average case (success): average number of access to f for successful search**
- **average case (failure): average number of access to f for unsuccessful search**
- **worst case: maximal number of access to f for successful search**

Sequential Search

Algorithm:

1. If f is empty, the search is unsuccessful.
2. If f[istart]=s, the search is successful.
3. Else: perform sequential search in the index set [istart+1,iend].

Complexity of Sequential Search:

Worst case: n
Average Case (success): 1/n * sum(i:i=1,...,n) = (n+1)/2
Average Case (failure): n

Remarks:

If f is sorted, then the average case complexity in the case of failure is comparable with the average case complexity in the case of success

The following algorithms are based on the assumption that f is sorted

Block Search

Pre Condition:
- **f is sorted (ascending)**
- **the index set is divided into n/t blocks of size s**

Algorithm:

1. In the sequence of the last entries of each block (having the maximal value in the block), search for the index of the first key which is greater than or equal to s.
2. If this search is unsuccessful, the the whole search is unsuccessful.
3. Else: Perform sequential search in the found block.

Complexity of Block Search:

Worst case: n/t+t
Average Case (success): t/n * sum(i:i=1,...,n/t) + (t+1)/2 = (n/t + t)/2 +1
Average Case (failure): comparable

Binary Search

Algorithm:

1. If f is empty, the search is unsuccessful
2. Compare s with the sequence entry in the mid of the sequence (index imid)=(istart+iend)/2:
3. If both are equal, the search is successful.
4. If s is smaller than f[imid], then perform binary search in the index set [istart,imid-1].
5. If s is greater than f[imid], then perform binary search in the index set [imid+1,iend].

Complexity of Binary Search:

When performing a successful binary search with n=2k-1, one finds**

1 key with 1 comparison
2 keys with 2 comparisons
4 keys with 3 comparisons
2(k-1) with k=log(n+1) comparisons**

This gives the following results:

Worst case: log(n+1)
Average Case (success): 1/n * sum(2**(i-1)*i:i=1,...,k) = (n+1)/n * log(n+1) -1
Average Case (failure): log(n+1)

Remarks:

1. **Binary Search is the optimal search algorithm based on comparisons if no additional assumptions are made.**
2. **Interpolation Search is a modification of binary search where the search is not divided into two equal parts but by interpolation of the search key in the between first and last key.**

Hashing

Basics:

The value of the key determines the position in a hash table.
The function which maps the key to the position is called hash function.
If two keys are mapped to the same position, this is called a collision.
A technique for resolving collisions is needed.

Special case: If the key values are in some reasonable , the hash function can be choosen as the identity. This technique is called table lookup.

Algorithm (insert):

1. Create an empty hash table with N >n entries.
2. Für all keys s, perform steps 3 to 5:
3. Compute the hash function for the argument s as insert position
4. If this position is not empty, search for an alternative insert position according to a given collision resolution strategy
5. Insert the data at the insert position

Algorithm (search):

1. Compute the hash function for the given key s
2. Beginning at this position and following the given collision resolution strategy, search for the occurrence of the given key.
3. The result of this search determines the result of the search for the given key.

Hash Functions

Requirements:

1. Deterministic
2. Simple
3. Scattering keys over the interval

Examples:

1. If the key can be represented as a machine word, take b bits from the middle of the key to map the key into the [0,2**b-1]
2. If the key can be represented as an integer, define the hash function as hash(s)=s mod N, where N is prime
3. If the key consists of several machine words, the machine words are mapped to one machine word (exclusive or, folding techniques, …)
4. If there is some information on the distribution of the keys, it makes sense to work with a specific hash function

Requirements of a Collision Resolution Strategie:

Every entry in the hash table must be a possible target!

Collision Resolution Strategies

1. Linear incrementing

I(k) = (I(k-1) +h) mod N, k=1,…,N-1, with a natural number h

If h is relatively prime to N, then every position can be reached

Disadvantage: primary clustering

2. Random hashing

Let (p(1),…,p(N-1)) be a permutation of 1,…,N-1 and

I(k) = (I(0)+p(k)) mod N, I=1,…,N-1

Every position is reachable

Disadvantage: secondary clustering

3. Double hashing

Double hashing is a modification of the technique of linear incrementing where the increment is based on the key value.

Example:

H := max ((s div N) mod N,1)

4. Chaining

This technique is characterized by the fact that the alternate position of a key is not calculated, but the first free position is taken.

We distinguish between the case where the where the first free poisition is determined as soon as a collision occurs

- ⇨ coalesced chaining

and the case where the collisions are resolved in a second phase

- ⇨ separate chaining

This technique is not applicable in a dynamic environment.

In this case, keys which suffer a collision are usually organized in a linked list per hash table entry.

Analysis:

Worst case complexity: n

Average case complexity for uniform hashing (e.g., double hashing)

Consider we have N elements and B entries in the hash table (often called buckets)

Then the probability of a collision is
N/B in the first probe
N*(N-1)/B*(B-1) in the second probe
…
N*(N-1)*…*(N-I+1) / B*(B-1)*…*(B-I+1) in the I-th proble

Thus the average case complexity for unsuccessful search is (B+1)/(B+1-N) which is approximately 1/(1-f), where f is the fillfactor N/B.

The average case complexity for successful search is the average cost of all insertions made so far and thus approximately –(1/f)*ln(1-f).

Sorting

Specification

Pre Condition:
- **Array f of mutually distinct keys with index set [istart, ..., iend]**

Post Condition:
- **Permutation of f which ist sorted (ascending)**

Possible additional requirement:
- **Stabilty of Sorting Algorithm: Conservation of the relative position of identical keys**

Measures of Complexity:

Time Complexity (comparisons (C) vs. exchange (E) operations)
- **Worst case**
- **Average case**

Space Complexity
- **in situ algorithms**
- **ex situ algorithms**

Selection Sort

Algorithm (ex situ):

1. Initialize the target sequence as empty.
2. As long as the source sequence is not empty, perform the follwing steps:
3. Search for the minimum of the source sequence.
4. Insert the minimum at the end of the target sequence.
5. Delete the minimum from the source sequence.

This algorithm is reasonable, correctness and stabiliy are evident, but the implementation of deleting the minimum from the source sequence is not trivial. On the other hand, if the algorithm is formulated in situ, then the construction of the target sequence and the deletion from the source sequence is solved by exchanging the key at the present position with the minimum key in the remaining source sequence.

Algorithm (in situ):

1. If istart < iend, perform the following steps:
2. Search for the first index imin of the minimal key of f
3. Exchange f[istart] and f[imin]
4. Perform this algorithm in the [istart+1,iend]

Analysis:

The time complexity of the algorithm is C*n*(n-1)/2 + (n-1)*E (average case, worst case).
The number of comparisons is independent of the input sequence!

Stability:

If the algorithm is performed ex situ, then it is stable, if it is performed in situ, stability is lost.

Bubble Sort (Sorting by Exchange)

Algorithm:

1. If istart < iend, perform the following steps:
2. Starting with iend and finishing with istart+1, perform step 3:
3. If f[i] < f[i-1], exchange these sequence entries
4. Perform this algorithm in the [istart+1,iend]

Correctness and Stability:

Again, the correctness of the algorithm is evident. Stability results from the fact that exchanges are only made if f[i] < f[i-1].

Analysis:

The time complexity of the algorithm is

- **0.5*(C*(n*(n-1) + E*(n-1)*n) (worst case)**
- **0.5*C*n*(n-1) + 0.25*E*(n-1)*n (average case)**

The number of comparisons is independent of the input sequence. (But see the modifications discussed below.)

Modifications:

In general, bubble sort is the worst algorithm for sorting by comparison and exchange (because of the locality of the exchange operations!). Sometimes, "improvements" of bubble sort are discussed (e.g., stop when no exchanges have been made in the last iteration, shakersort), but these modifications do only help in special cases (sorted sequence, sequence sorted in reverse order) so that we will not discuss them in detail.

Insertion Sort

Algorithm:

1. Start with the sorted sequence containing only f[istart].
2. For f[i], i=istart+1,…,iend, perform step 3:
3. Insert f[i] into the sorted sequence f[istart], … , f[i-1] so that the resulting sequence is sorted.

Refinement of step 3:

3.1 Search for the correct insert position iinsert of f[i] in the istart,…, i-1.
3.2 Shift the sequence entries in the iinsert+1,…, i-1 one position to the right.
3.3 Insert f[i] at the insert position iiinsert.

Steps 3.1 and 3.2 can be performed simultaneously if the search is performed as a sequential search.

Correctness and Stability:

Again, correctness of the algorithm is evident. Stability is obvious, too, if the search for the correct insert position and the shift are performed simultaneously. If these operations are done in two separate steps, then the maximum of the possible insert positions must be used.

Analysis:

The time complexity of the algorithm is

- **(C+E)*(n*(n-1)/2) (worst case)**
- **about half this value in the average case**

The complexity of the algorithm is strongly dependant on the input sequence.

Remarks:

The idea to perform a binary search in step 3.1 seems to be obvious, but this does not really help as the number of exchange operations is not reduced.

Sorting by insertion is an algorithm which is appropriate for nearly sorted sequences. This fact is used in some algorithms, e.g., the shell sort algorithm.

Shell Sort

The idea of the algorithm is to apply insertion sort in subsequences with constant index distances repeatedly, first in small sub sequences with large index distances (where the quadratic complexity of insertion sort is not really a problem), then with decreasing index distances and finally with index distance 1. Shell sort relies on the fact that in the last iteration the sequence is nearly sorted so that insertion sort is a good algorithm to apply.

As experience has shown, starting with an index distance of 2k –1, where k = floor(log(n)), and dividing the index distance by 2 in every iteration is a good choice.**

Algorithm:

1. Start with an appropriate index sequence h (see above).
2. While h>0, perform the following steps:
3. For i=1,…,h, perform insertion sort in the subsequence in the interval i,i+h,…
4. Divide h by 2.

Correctness and Stability:

As the last step of shell sort is an insertion sort, correctness is evident. But although the basic algorithm insertion sort is stable, shell sort is not stable.

Analysis:

The analysis of shell sort is quite complicated. One can show that the worst case complexity can be estimated by O(n^1.5) and the average case complexity can be estimated by O(n^1.2).

Quick Sort

Algorithm (Basic formulation):

1. If istart < iend, perform the following steps:
2. Take f[istart] as separating value.
3. Construct a partition of the sequence in the interval istart+1,…, iend and an index ipart so that
   f[i] < f[istart] for all i in istart+1,…,ipart
   f[i] >= f[istart] for all i in ipart+1,…,iend.
4. Exchange f[istart] and f[ipart].
5. Perform the algorithm in the interval istart,…, ipart-1.
6. Perform the algorithm in the interval ipart+1,…, iend.

The construction of the partition in step 3 can be performed by the following algorithm which results in a partition of f so that some predicate P is true for all i in istart, …, ipart whereas P is false for all i in istart+1, …, iend.

1. **Initialize ipart with istart-1.**
2. **If istart > iend, the we are ready.**
3. **If istart = iend, perform step 4:**
4. **If P is true for f[istart], set ipart=istart.**
5. **Else perform steps 6 to 9:**
6. **Starting with ipart, search for the first index i1 so that P is false for f[i1].**
7. **Starting with iend, search for the first index i2 so that P is true for f[i2].**
8. **If i1 < i2, exchange f[i1] and f[i2], increment i1 and decrement i2 by 1.**
9. **Perform the algorithm in the interval i1,…, i2.**

Correctness and Stability:

- **As termination of quick sort is evident, the correctness of quick sort depends on the correctness of the construction of the partition in step 3. Quick Sort is not stable.**

Analysis:

Worst case (sorted sorted sequence!): T(n)=C*n + T(n-1), n>1, T(1)=0
=> T(n) = C* (n*(n+1)/2 – 1), n>=1

Best Case: T(n)=C*n + T(n/2), T(1)=0
=> T(n) = C*n*log(n), n>=1

Average case: T(n)=(C+E/6)*2*ln(2)*n*log(n) = O(n*log(n))

Modifications:

- **Take the median of f[istart], f[iend], f[(istart+iend)/2] as separating value. This does not prevent the worst case complexity of O(n^2), but the probability for this situation is reduced. Moreover, sequences which are sorted or sorted in reverse order are treated best.**
- **To reduce the stack requirements of quick sort, the smaller sequence should be sorted before the larger sequence.**
- **To avoid recursion, one may implement an iterative version of quick sort by using the abstract data type stack.**
- **As insertion sort is a very good algorithm for nearly sorted sequences, one modification of quick sort is to stop recursion if the size of the sequences to sort becomes reasonable small (suggestion: 20-25) and perform insertion sort at the end of the process.**
- **Quick sort is an algorithm which cannot only be used for sorting internal sequences, but the idea can also be adapted to sort external random access data structures in situ!**

The following modification of quick sort can be used to find the k-th smallest element from an unsorted sequence:

Let K=istart-1+k.

1. **If istart<=K<=iend, perform the following steps:**
2. **Take the first element of the sequence as separating value.**
3. **Construct a partition of the sequence in the interval istart+1,…, iend and an index ipart so that**
   f[i] < f[istart] for all i in istart+1,…,ipart
   f[i] >= f[istart] for all i in ipart+1,…,iend.
4. **Exchange f[istart] and f[ipart].**
5. **If ipart=K, then we are ready.**
6. **If ipart < K, perform the algorithm in the interval ipart+1,…, iend.**
7. **If ipart > K, perform the algorithm in the interval istart, …, ipart-1.**

Merge Sort

Algorithm:

1. If the sequence is empty or consists of one element, processing is finished, else perform the following steps:
2. Construct a partition of f where f1 and f2 are of (nearly) equal size.
3. Sort f1 and f2.
4. Merge f1 and f2 to construct the sorted sequence.

The construction of the sequence in step 3 is easier than in the case of quick sort: take ipart=(istart+iend)/2 as separating index and copy f[istart], ..., f[ipart] into f1 and f[ipart+1],..., f[iend] into f2.

The merge algorithm in step 4 can be performed as follows:

1. **Initialize the resulting sequence f as empty.**
2. **As long as neither processing of f1 is finished nor processing of f2, perform steps 3 and 4:**
3. **If the current element of f1 is smaller or equal the corresponding element of f2, append it to f and increment the current position of f1 by 1.**
4. **Else append the current element of f2 to f and increment the current position of f2 by 1.**
5. **Append the remainder of f1 to f.**
6. **Append the remainder of f2 to f.**

Correctness and Stability:

Correctness and stability of merge sort are evident.

Analysis:

Space complexity of merge sort is 2*n!
Time complexity of merge sort is

- **T(n)=2*(C+E)*n + T(n/2)) => T(n) = 2*(C+E)*n*log(n), n>=1 (T(1)=0!)**

Iterative version of merge sort:

1. Initialize l by 1.
2. Construct a partition of f into (nearly) equal sequences f1 and f2.
3. Choose f1 and f2 as sources.
4. While l is smaller than the size of f, perform steps 5 to 7:
5. Merge subsequences of length l from both sources into sorted subsequences of length 2*l and append them to target 1 and target 2.
6. Let l=2*l.
7. Exchange the roles of source and target sequences.
8. The first source sequence is the resulting sorted sequence.

Correctness and Stability:

Correctness and stability are evident.

Analysis:

Space complexity is 2*n.
Time Complexity is (C+E)*n*(1+log(n)), n>=1.

Remarks:

The iterative version of merge sort is based only on sequential access to a sequence. Hence, merge sort is an algorithm which is absolutely appropriate for sorting sequential files.

We will see that the complexity of sort algorithms based on comparisons and exchanges cannot be improved beyond O(n*log(n)) in general.

Next we will have a look at some sort algorithms which have a better asymptotic(!) behaviour.

Sorting by counting

The algorithm assumes that the keys are contained in some area min, …, max which is an appropriate index area for a sequence. In this sequence called count we count how many keys in the original sequence have a value equal to the current index.

Algorithm:

1. Initialize the entries in the sequence count by 0.
2. During a sequential scan of f, perform step 3 for all indexes i:
3. Increment count[f[i]] by 1.
4. Calculate pos[i], i=istart, …, iend, as follows:
   pos[min]=istart,
   pos[i]= pos[i-1]+count[i], i=min+1,..., max.
5. During a sequential scan of f, perform steps 6 and 7:
6. Insert f[i] at position pos[f[i]] in sequence g.
7. Increment pos[f[i]] by 1.
8. g is the resulting sorted sequence.

Correctness and Stability:

Correctness and stability of the algorithm are obvious.

Analysis:

Time complexity of the algorithm is c1*n+c2*m with m=max-min+1 and thus O(n).
Space complexity of the algorithm is 2*n+2*m but can be improved.

Bottomp up radix sort

The idea of the algorithm is to use sorting by counting repeatedly to sort a sequence of keys which are considered as byte pattern (e.g., strings, but integers can be considered, too). From the least to the most significant byte, sorting by counting is applied with the current byte considered as key.

Algorithm:

1. Consider the keys as divided into d parts of cardinality m. (m=256 => division in bytes, m=2 => division in bits)
2. From the least significant part to the most significant one, apply sorting by counting where the value of the current part is considered as key.

Correctness and Stability:

Correctness and stability depend on the fact that sorting by counting is stable. This is true for the ex situ version discussed on the previous slide.

Analysis:

Time complexity of the algorithm is linear, but if the size of the pattern is too big, quick sort is superior. If the size is small, however (e.g., when the keys are integers and considered as string of four bytes), then the algorithm is better than quick sort.

Remark**:**

Bottom up radix sort can be combined with insertion sort, e.g., only the most significant parts of the key are considered for bottom up radix sort, the final sort is performed as insertion sort.

Interpolation sort

The idea of the algorithm is to calculate the position of a key by its value. (The idea can be compared to the idea of hashing!)
To achieve this goal, we need a function which maps the keys into the index area which is “nearly” strictly monotone increasing.

We consider the case where the index area is 0,..., N-1 with N>n so that the probability of collisions is reduced. The entries in the resulting sequence are called buckets as in the case of a hash table.

One popular function is the interpolation of the sequence values with minimum value min and maximum value max in the interval 0,..., N-1:

F(s) = ((s-min)*(N-1)) / (max-min)

As in the case of hash algorithms, a strategy for collision resolution is needed. One strategy is to keep one sorted list for every bucket.

Alternatively, interpolation is only used to distribute the keys over the buckets. Afterwards, if the number of entries in a bucket is small, insertion sort is used for sorting, else mergesort is used. (Insertion sort and merge sort are appropriate sorting algorithms for lists!)

Lists

Representations:

There are several representations of a linked list which may be appropriate for certain applications:

- **Unordered vs. Ordered lists**
- **Simply vs. Doubly linked lists**
- **Linear vs. Circular lists**
- **Lists with and without dummy element**

These are 16 combinations. We will not discuss all of them and consider circular doubly linked lists with dummy element. We will distinguish between the representation as unordered and ordered list.

Each list node consists of a key, a forward link to its successor and a backward link to its predecessor. As we consider circular lists, the forward link of the last node refers to the first node, the backward link of the first node refers to the last node.

The list itself is represented by a reference to itsdummy node. An empty list is represented by the fact that the dummy node is (doubly) linked to itself.

For unordered lists, we will consider five basic operations:

Insert (at the beginning, at the end, after some position)
Delete (fom the beginning, from the end, at some position)
Empty
GetKey(AndData) (first node, last node, node at some position)
Search ((first) position from key)

For ordered lists, we will consider three basic operations:
Insert (so that the list remains sorted)
Delete
Search

Unordered lists:

Insert a key s at the beginning of the list (after the dummy node):

1. **Make a new node N.**
2. **Set the key value of N to s.**
3. **Set the forward link of N to the successor of the dummy node.**
4. **Set the backward link of N to the dummy node.**
5. **Set the backward link of the former successor of the dummy node to the new node.**
6. **Set the forward link of the dummy node to the new node.**

This algorithm can be applied to insert any node after some node with known position in the list.

Insert a key s at the end of the list (before the dummy node):

1. **Make a new node N.**
2. **Set the key value of N to s.**
3. **Set the forward link of N to the dummy node.**
4. **Set the backward link of N to the predecessor of the dummy node.**
5. **Set the forward link of the former predecessor of the dummy node to the new node.**
6. **Set the backward link of the dummy node to the new node.**

Empty:

1. **Look if the target of the list reference is linked to itself.**

Delete the first node N from the list (if the list is not empty):

1. **Set the forward link of the predecessor of N to the successor of N.**
2. **Set the backward link of the successor of N to the predecessor of N.**
3. **Delete node N.**

The same algorithm can be applied to delete any node in the list given its position.

Getting the key (and data) of some node from the list given its position is trivial. As the list is represented by a reference to the dummy node, the problem of getting the key (and data) of the first node and the last node is solved because the first node is the successor of the dummy node and the last node is the predecessor of the dummy node.

Search for the (first) position of a key s:

The search for a key has to be performed as a sequential search. The characteristics of the sequential search may be improved by using a self organizing list: In the case of a successful search, the node where the key has been found is moved to the top of the list.

Ordered lists:

Insert a key s:

1. **Search for the position of the first node P having a key greater than s.**
2. **If the search is successful, perform steps 3 to 10:**
3. **Make a new node N.**
4. **Set the key value of N to s.**
5. **Set the forward link of N to a reference to P.**
6. **Set the backward link of N to a reference to the predecessor of P.**
7. **Set the forward link of the predecessor of P to N.**
8. **Set the backward link of P to N.**
9. **If P is the beginning of the list, perform step 10:**
10. **Let the list be represented as reference to the new node.**
11. **If the search is unsuccessful, append the key to the ordered list by performing steps 3 to 8 with P representing the first node of the list.**

Delete a key s:

1. **Search for the position of key s.**
2. **Perform the deletion of the corresponding node for which the position is known.**

Search for a key s:

The search for a key has to be performed as a sequential search, too, but one can stop if the search key is greater than the current key in the sequence.

Abstract Data Types

Stack

- **insert (push)**
- **return last inserted object (top)**
- **delete last inserted object (pop) => LIFO**
- **check if stack is empty**

Stacks can be implemented as unordered lists where insertion and deletion is performed at the beginning.

Queue

- **insert (enqueue)**
- **return first inserted object (front)**
- **return last inserted object (back)**
- **delete first inserted object (dequeue) => FIFO**
- **check if queue is empty**

Queues can be implemented as unordered lists where insertion is performed at the end and deletion is performed at the beginning.

Set

- **insert**
- **search**
- **delete**
- **check if set is empty**

- **union**
- **intersection**
- **difference**

Sets can be implemented as ordered lists without duplicates.
Additional algorithms are needed for union, intersection and difference.

Algorithm for union:

1. **Let s1 and s2 be the first and second set (=ordered list), respectively.**
2. **Let r be the resulting list.**
3. **While processing of s1 is not finished and processing of s2 is not finished, perform steps 4 to 13.**
4. **If the current keys of s1 and s2 are equal, perform steps 5 to 7:**
5. **Append the current key of s1 to r.**
6. **Proceed one position in s1.**
7. **Proceed one position in s2.**
8. **If the current key of s1 is smaller than the current key of s2, perform steps 9 and 10:**
9. **Append the current key of s1 to r.**
10. **Proceed one position in s1.**
11. **If the current key of s2 is smaller than the current key of s1, perform steps 12 and 13:**
12. **Append the current key of s2 to r.**
13. **Proceed one position in s2.**
14. **While processing of s1 is not finished, perform steps 15 and 16:**
15. **Append the current key of s1 to r.**
16. **Proceed one position in s1.**
17. **While processing of s1 is not finished, perform steps 18 and 19:**
18. **Append the current key of s1 to r.**
19. **Proceed one position in s1.**

The algorithms for intersection and difference are similar and will not be discussed in detail.

Open Hashing:

If hashing is considered in a dynamic environment, the techniques of collision resolution subsumed under the notion of closed hashing (collision resolution within the hash table) are not appropriate.
In this situation, the usual way to perform collision resolution is to keep one container per bucket where all the keys mapped to the bucket are held.
If the hash table is used to represent sets (no duplicates allowed), the the container should be implemented using the abstract data type set, if duplicates are allowed, the container should be implemented using an unordered list.

Sorting Algorithms for linked lists:

1. **Selection sort uses only the linearity of the sequence.**
2. **Bubble Sort uses the doubly linked list structure.**
3. **Insertion sort uses only the linearity of the sequence. The insertion is even simpler in the linked list implementation because no shifts are necessary!**
4. **Shell sort uses the concept of index distances. Therefore it is not appropriate for linked lists.**
5. **Quick Sort can be implemented for singly linked list but the more efficient variants need the doubly linked list structure.**
6. **Merge sort uses only the linearity of the data structure.**
7. **Sorting by counting and bottom up radix sort use only the linearity of the sequence.**
8. **Bucket sort is appropriate for linked lists as it is the combination of calculating the bucket from the key value, thus distributing the sequence over the buckets, and then applying some sort algorithm appropriate for linked lists to the keys contained in one bucket (e.g., insertion sort for small lists and merge sort for large lists).**
   It can be viewed as the counterpart of open hashing.