Лекция 10.

Поиск

Постановка задачи поиска

- Пусть R_1, R_2, ..., R_n – множество записей
- K_1, K_2, … , K_n – множество ключей

В алгоритмах поиска обычно задан аргумент поиска K, и задача состоит в нахождении записи, имеющей K своим ключом. Существуют две возможности окончания поиска: либо поиск оказался удачным, т.е. была найдена запись с ключом K, либо он оказался неудачным, т.е. ключ K не был найден ни в одной записи.

Поиском называется процедура выделения из множества записей его подмножества, записи которого удовлетворяют заданному условию. Множество записей может храниться в массиве, связанном списке, дереве или графе.

- Найти в таблице запись с заданным ключом ***K;***
- или установить, что такого ключа в таблице нет;
- если возможно, то вставить в таблицу новую запись, содержащую ключ ***K.***

Ключ	Данные
K_1	
K_2	
K_3	
• • •	
K_n	

Эффективность алгоритмов поиска

Во многих задачах поиск требует наибольших временных затрат, поэтому замена «плохого» метода поиска на «хороший» часто ведет к существенному увеличению скорости работы. Нередко удается так организовать данные, что поиск значительно убыстряется. Эффективность различных алгоритмов поиска оценивается количеством сравниваемых пар ключей, необходимым для выполнения условия поиска. Обычно оценивается среднее и максимальное количество сравнений.

Последовательный поиск

Алгоритм 1:

1. Установить **i = 1**.
2. Если **K = K_i** то УДАЧА.
3. Иначе **i = i + 1**.
4. Если **i ≤ N**, то вернуться к шагу 2
5. Иначе НЕУДАЧА.

Условия окончания поиска :

1. Элемент найден, т.е. обнаружен ключ Ki=K.
2. Все записи просмотрены, и совпадение не обнаружено.

Максимальное число сравнений при удачном и неудачном поиске равно N. Среднее число сравнений при удачном поиске равно (N+1)/2. Для этого метода поиска, как среднее, так и максимальное время поиска пропорционально N, т.е. сложность рассмотренного алгоритма равна O(N).

Имеются две проверки: на совпадение ключей (K = Ki) и на выход значения индекса за пределы массива(i <= N).

Быстрый последовательный поиск

Если гарантировать, что ключ всегда будет найден, то без второго условия (выход индекса за границу массива) можно обойтись. Достаточно в конце массива поместить дополнительный N+1-й элемент, присвоив ему значение искомого ключа **K**. Назовем такой элемент барьером, так как он предотвращает выход за пределы массива.

Ключ будет найден в записи с индексом i.

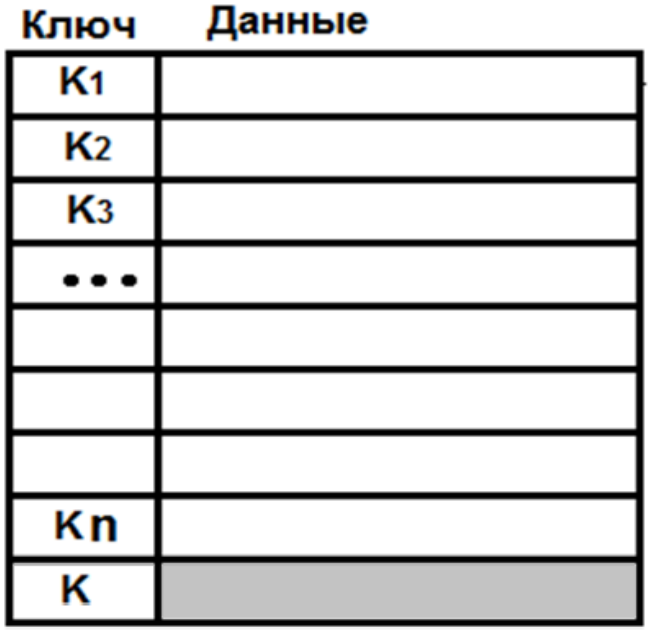
При i = N+1 (конец массива) ключ не найден (т.е. найден только в барьере). Поиск неудачен.

В противном случае - поиск удачен. Такой подход позволяет исключить в среднем (N + 1)/2 сравнений.

Быстрый последовательный поиск

Алгоритм 2.

1. Установить i = 1 и K_{N+1} = K.
2. Если K = Ki, то перейти на 4.
3. Увеличить i на 1 и вернуться к шагу 2.
4. Если i ≤ N то УДАЧА, иначе НЕУДАЧА (i = N + 1).



- Предполагается, что ключ Ki будет разыскиваться с вероятностью Pi, где

$$P1 + P2 + \cdots + P_N = 1$$

- Среднее время удачного поиска

$$T_{cp} \approx 1*P1 + 2*P2 + \cdots + N*P_N$$

- T_{cp} минимально при

$$P1 \geq P2 \geq \ldots \geq P_N$$

Последовательный поиск в упорядоченной таблице

K – искомый ключ. Ключи $K_1 < K_2 <... K_n$.

Необходимо найти запись с заданным ключом K. В отличие от поиска в неупорядоченном массиве, просмотр элементов упорядоченного массива заканчивается в тот момент, когда первый раз выполнится условие K ≤ Ki. Очевидно, что все элементы Ki+1,...,Kn будут больше.

Среднее число сравнений и при удачном, и при неудачном поиске в алгоритме последовательного поиска в упорядоченном массиве равно (N+1)/2.

Последовательный поиск в упорядоченной таблице

Алгоритм 3.

К – искомый ключ. Ключи $\mathbf{K_1 < K_2 <... K_N}$.

1. Установить i = 1.
2. Если К ≤ K_i, то перейти на 4.
3. Увеличить i на 1 и вернуться к шагу 2.
4. Если К = Кi, то УДАЧА, иначе НЕУДАЧА и нужной записи в таблице нет.

Методы поиска в таблицах со случайным доступом и упорядоченными ключами $\mathbf{K_1 < K_2 < ... < K_N}$.

После сравнения К и Кi :

- если К < Кi, то Кi, Кi+1, ..., K_N исключаются из рассмотрения;
- если К = Кi, то поиск закончен;
- если К > Кi, то K_1, K_2, ..., Кi исключаются из рассмотрения.

Бинарный поиск

- Использует два указателя: **l** и **u**, соответствующие верхней и нижней границам поиска

Эффективность алгоритма

- $T_{ср} \approx \log_2 N$ - число сравнений

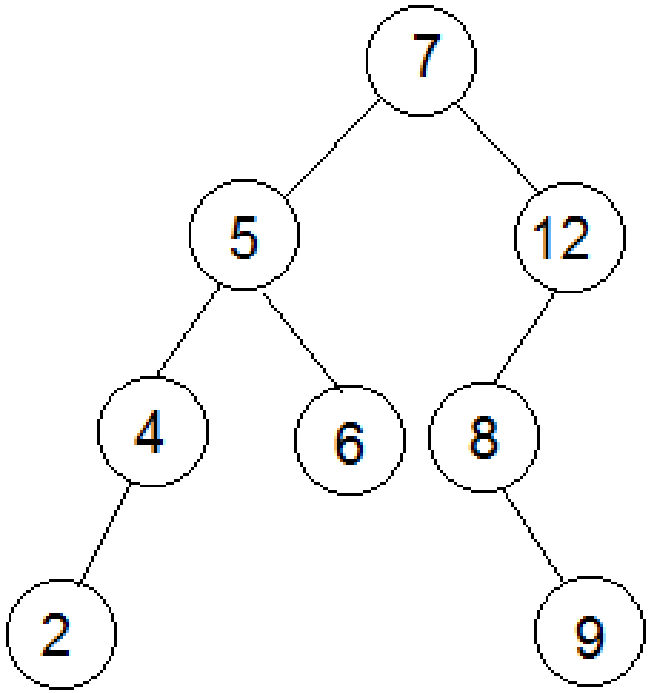
Бинарный поиск. Алгоритм

1. Установить $l \leftarrow 1$, $u \leftarrow N$.
2. Если $u<l$, то НЕУДАЧА в противном случае установить $i \leftarrow (l + u)/2$
3. Если $K<Ki$, то перейти на 5; если $K>Ki$, то перейти на 6
4. если $K = Ki$ то УДАЧА
5. Установить $u \leftarrow i - 1$ и вернуться к шагу 2.
6. Установить $l \leftarrow i + 1$ и вернуться к шагу 2.

Поиск по бинарному дереву

- Понятие бинарного дерева поиска

7 12 5 4 8 2 6 9



Поиск со вставкой по дереву

- Производится поиск заданного ключа K. Если его нет в таблице, то в дерево вставляется новый узел, содержащий ключ K.
- Узлы дерева содержат поля:

  KEY(P) = ключ, хранящийся в узле NODE(P);

  LLINK(P) = указатель на левое поддерево узла NODE(P);

  RLINK(P) = указатель на правое поддерево узла NODE(P);
- Переменная ROOT указывает на корень дерева.

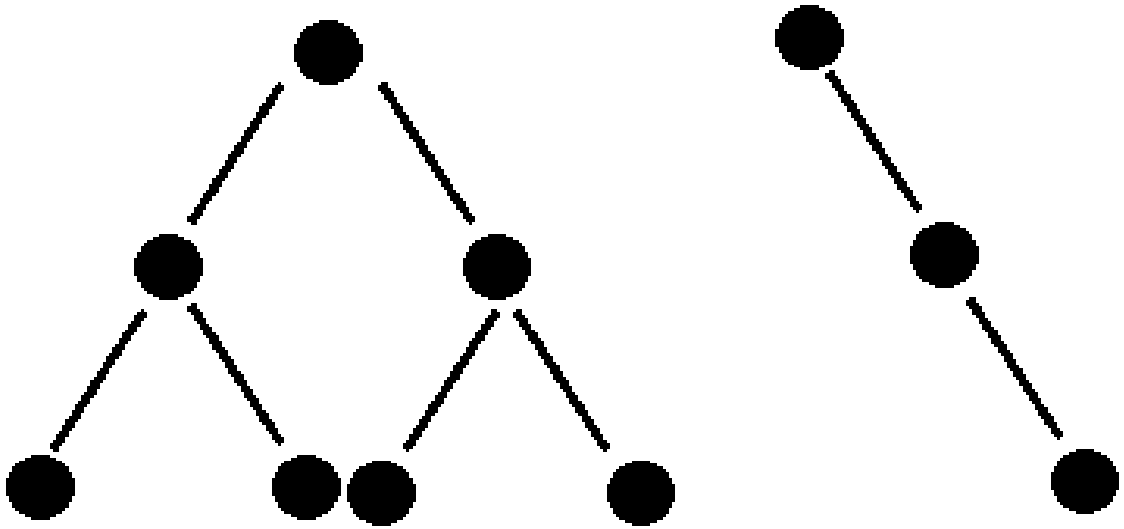
Поиск с вставкой по дереву. Алгоритм

1. Установить P = ROOT.
2. Если K < KEY(P), то перейти на T3; если K > KEY(P), то перейти на T4; если K = KEY(P) то УДАЧА.
3. Если LLINK(P) != NULL, установить P = LLINK(P) и перейти на 2, иначе перейти на 5.
4. Если RLINK(P) != NULL установить P = RLINK(P) и перейти на 2, иначе перейти на 5.
5. Создать новый узел Q. Установить KEY(Q) = K, LLINK(Q) = RLINK(Q) =NULL.

   Если K < KEY(P), установить LLINK(P) = Q иначе установить RLINK(P) = Q.

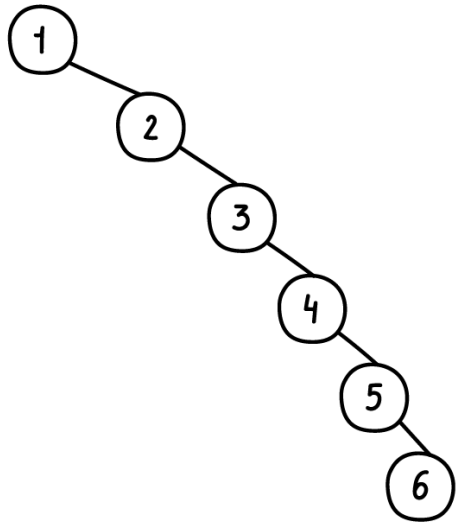
Эффективность алгоритма

- Для достаточно хорошо сбалансированного дерева время поиска пропорционально $\log_2 N$.
- Для вырожденного дерева — N.

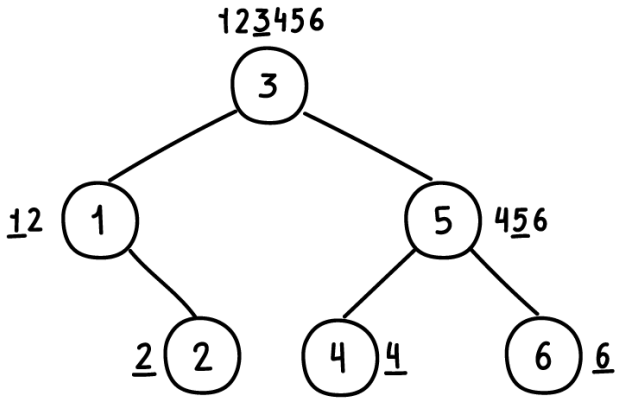


Сбалансированные деревья

Идеальная сбалансированность — это свойство дерева, при котором все его уровни, иногда кроме последнего, полностью заполнены. Для сравнения рассмотрим два бинарных дерева поиска, которые будут представлять собой одну последовательность элементов — 1,2,3,4,5,6. Из этой последовательности можно построить несколько деревьев, например:



(а)



(б)

В дереве (б) каждый из уровней, кроме левой ветки узла 1, заполнены. Значит, дерево (б) — идеально сбалансированное дерево. Что нельзя сказать о дереве (а). Дерево (а) и дерево (б) можно отнести к бинарным деревьям поиска. Но для худшего случая — поиска в дереве элемента №6 в дереве (а) — требуется выполнить шесть операций перехода между вершинами, а в случае (б) — только три.

Дерево с максимально возможной высотой (а) называется **разбалансированным** или **вырожденным деревом**. Оно не отличается от двусвязного списка, значит, теряет свое основное преимущество — скорость поиска.

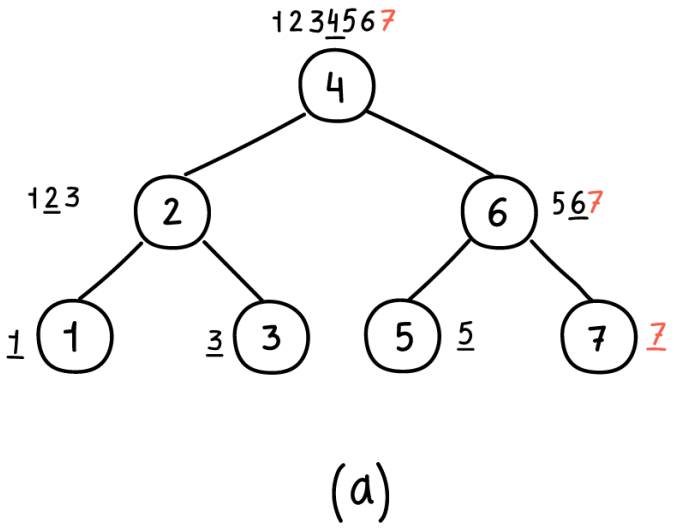
Для разбалансированных деревьев скорость поиска составляет **N**. А для идеально сбалансированного дерева (б) поиск будет завершен за число шагов, которые не превышают высоту дерева или за $\mathbf{\log_2 N}$.

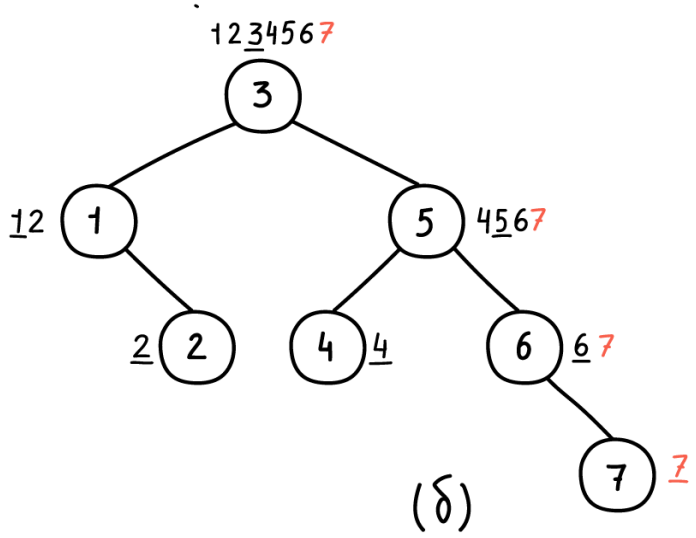
Сбалансированные деревья поиска

- Высота дерева $\log_2 N$
- Сбалансированное дерево: высота левого поддерева каждого узла отличается от высоты правого поддерева не больше чем на ±1.

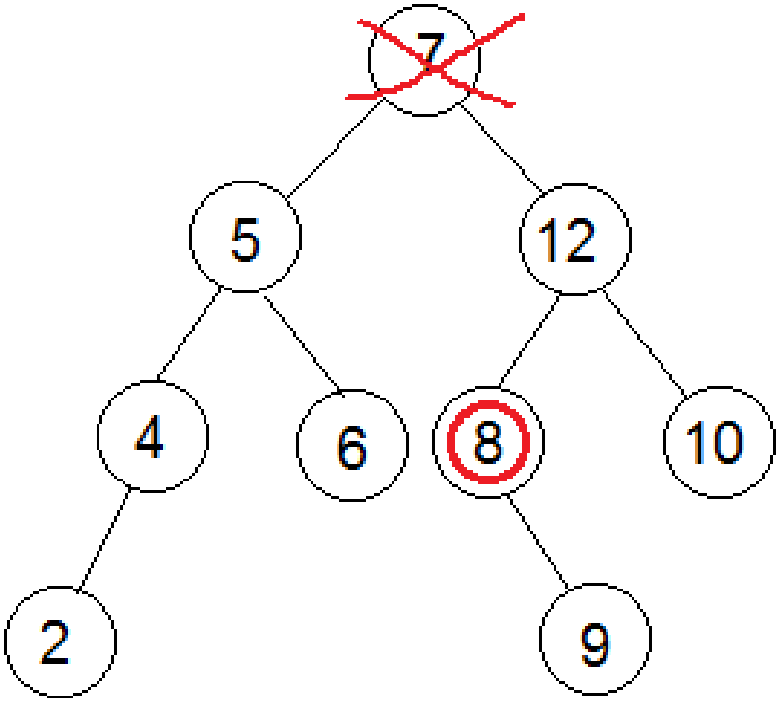
Состояние идеальной сбалансированности в дереве трудно поддерживать. Любое добавление или удаление узла в сбалансированном дереве может привести к ситуации, когда дерево выйдет из идеально сбалансированного состояния. В таком случае скорость поиска будет значительно деградировать после каждой вставки или удаления узла. Чтобы вернуть дерево в сбалансированный вид, его перестраивают после каждой манипуляции с составом узлов.

В случае (б) дерево вышло из состояния идеальной сбалансированности, так как оба нижних уровня не заполнены полностью. В таком случае поиск элемента №7 будет выполнен за четыре операции. В случае (а) элементы были перераспределены, чтобы сохранить сбалансированное состояние. В таком случае поиск элемента №7 будет выполнен всего за три операции.

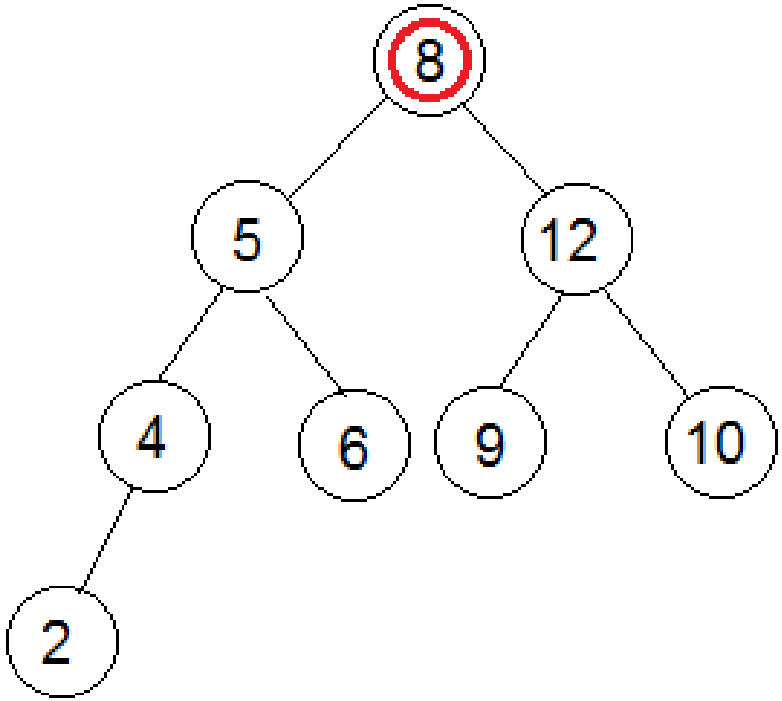




Удаления



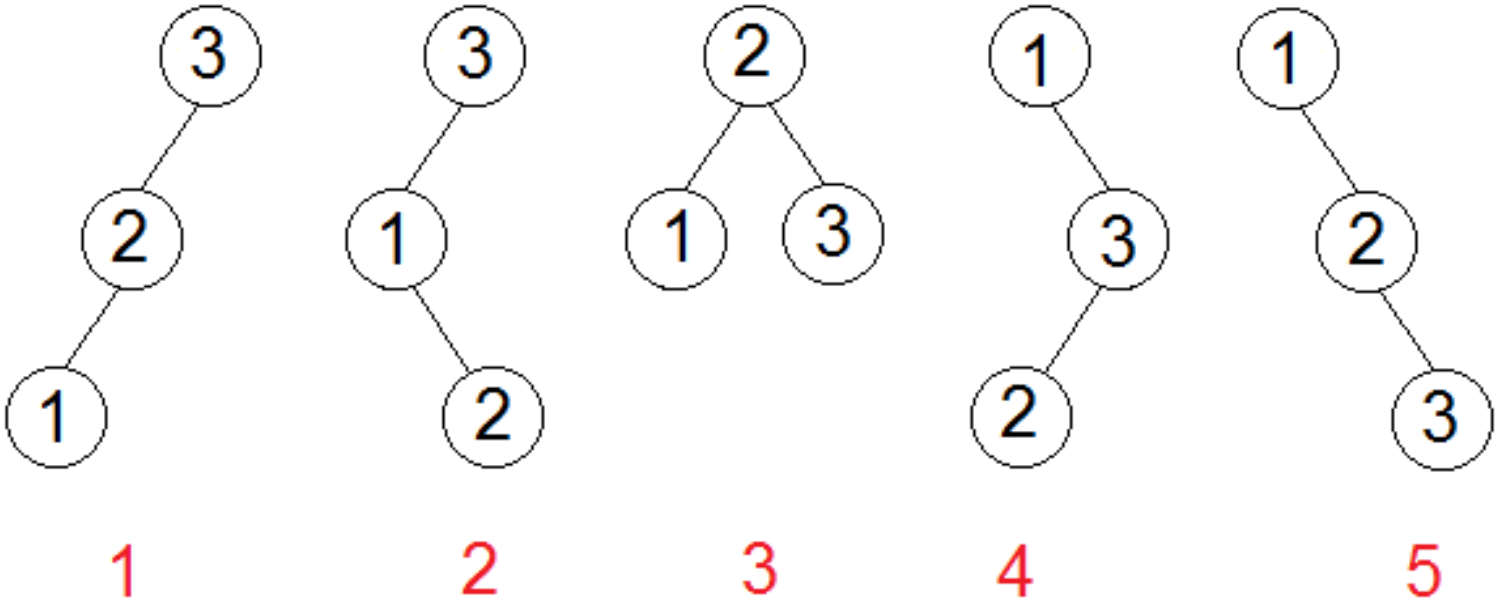
а)



б)

Оптимальные и сбалансированные деревья поиска

- Пусть даны три ключа: K1 < K2 < K3, K1=1 K2=2 K3=3
- Известны P1, P2, P3



Среднее время поиска

- P3 + 2P2 + 3P1
- P3 + 2P1 + 3P2
- P2 + 2P1 + 2P3
- P1 + 2P3 + 3P2
- P1 + 2P2 + 3P3

Требует порядка N^2 времени и памяти.

Самоорганизующийся список

Для ускорения поиска в несортированном массиве предлагается передвигать записи ближе к началу массива. Наиболее востребованные записи концентрируются в начале массива и поиск ускоряется в c/lnN раз, где N — размер массива, а c — константа, зависящая от используемой стратегии перемещения элементов.

Самая простая реализация самоорганизующегося списка - это связанный список, который эффективен при случайной вставке узлов и распределении памяти, и неэффективен при доступе к случайным узлам. Самоорганизующийся список снижает неэффективность за счет динамического переупорядочивания узлов в списке в зависимости от частоты доступа.

Самоорганизующийся список переупорядочивает узлы, оставляя наиболее часто используемые во начале списка. Шансы получить доступ к узлу, к которому ранее обращались много раз, выше, чем шансы получить доступ к узлу, к которому не обращались так часто. В результате размещение часто используемых узлов во главе списка приводит к сокращению количества сравнений. Это приводит к повышению эффективности и в целом сокращению времени выполнения запросов.