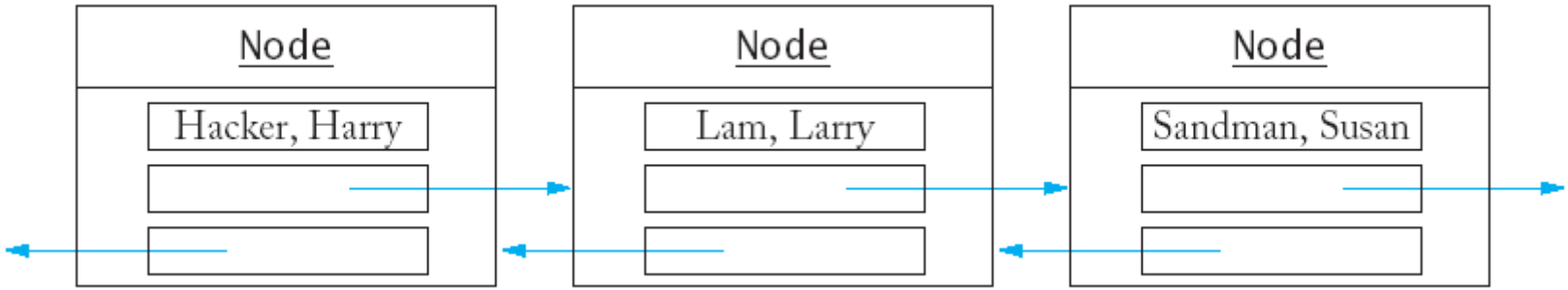
## 7. Въведение в структури от данни

*План:*
Свързани списъци - основни понятия
Свързан списък, стек и опашка в Стандартната библиотека шаблони (STL)
Други стандартни контейнери в STL
Алгоритми в STL

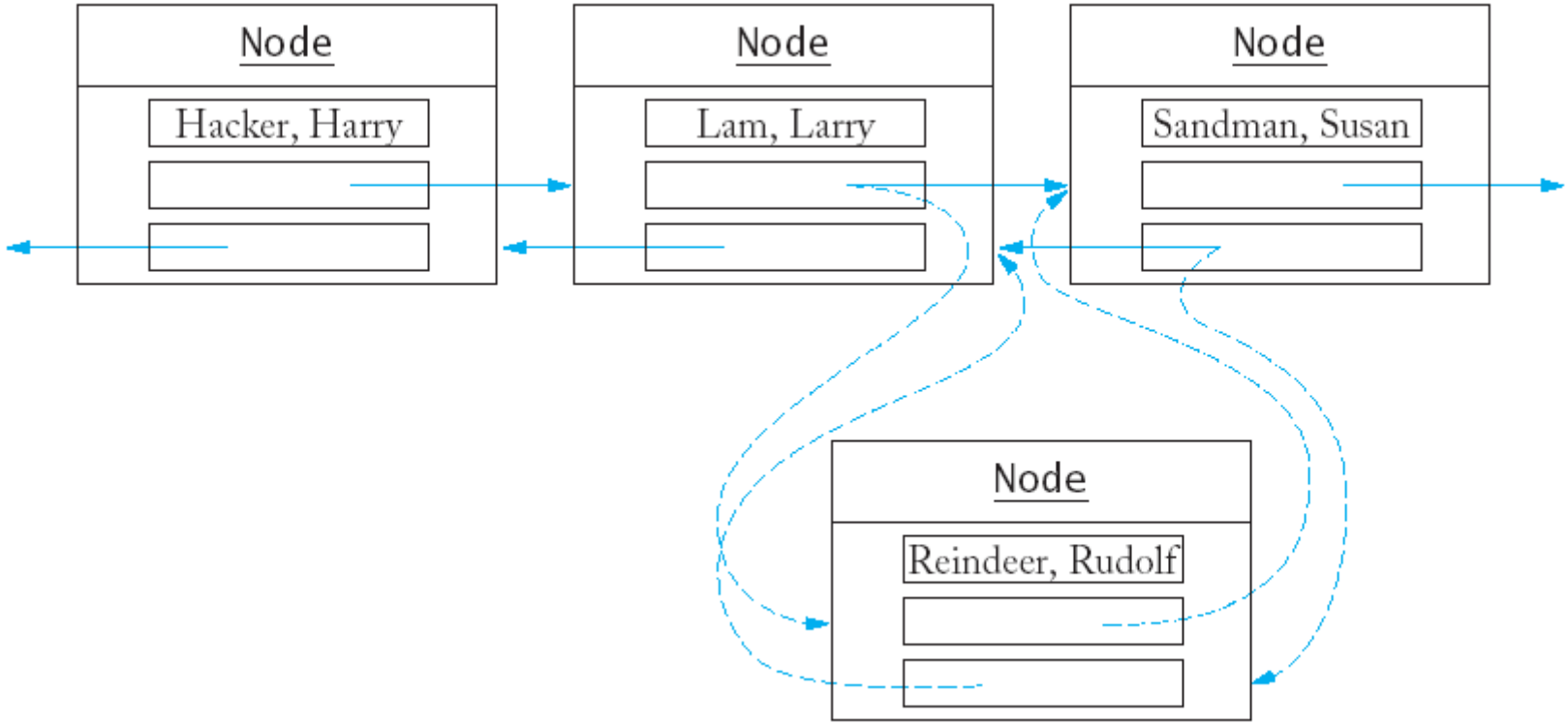
---

### Свързани списъци - основни понятия

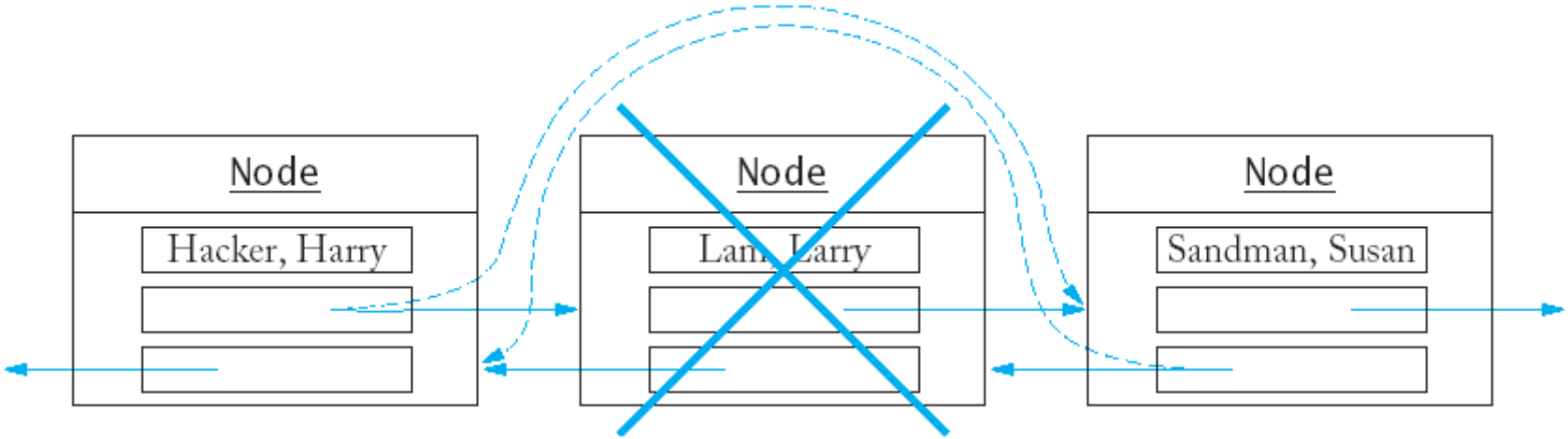
* Свързан списък е линейна структура от данни (редица), където всяка стойност се съхранява в отделен блок от паметта, заедно с местоположението на съседните блокове в редицата.
* Този начин на съхранение на елементите на редицата позволява лесно да се вмъкне или отстрани елемент, без да се местят другите елементи - сложност $O(1)$.
* Векторът (и масивът) е също линейна структура, но при вмъкване и изтриване на елемент се местят други елементи - сложност $O(n)$.
* Във всеки елемент на свързания списък се съхранява една данна (стойност) и две връзки (адреси) - към следващия елемент и към предишния елементи от списъка.




* Вмъкването и изтриването на елемент от свързан списък става лесно - променят се стойностите само на няколко връзки (адреси).
* На картинката:
- създава се нов елемент Rudolf;
- променя се следващия елемент след Larry (от Susan на Rudolf);
- променя се предишния елемент на Susan (от Larry на Rudolf); се
- добавя се предишен елемнт Larrry на новия Rudolf;
- добавя се следващ елемент Susan на новия елемнт Rudolf.




* Изтриването на елемент от свързан списък **не** изисква преместване на елементи на списъка.
* На картинката:
- променя се следващия елемент на Harry от Larry на Susan;
- променя се предишния елемент на Susan от Larry  на Harry;
- изтрива се Larry.




* За разлика от линейната структура вектор, където има пряк достъп до елементите му (с операция индекс), свързаният списък **не** осигурява пряк достъп (**не** поддържа операция индекс).
* За достъп до елементите на свързания списък е достатъчно да знаем местоположението (адреса) на първия елемент на списъка.
* Достъп до *k*-тия елемнт на свързан списък се реализизира, като се започне от първия му елемент и се обходят всички елементи до *k*-тия, като се броят посетените елементи - сложност $O(n)$.

---

** В стандартната библиотека шаблони (STL) на C++ има реализация на линейната структура свързан списък - шаблонът **`list`**.
* Всички основни операции на класа `list` се изпълняват за време $O(1)$.
* Достъп до елементите на свързания списък с $n$ елемента се осъществява посредством итератор за време $O(n)$.
*Пример:*

```
// list1.cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{  list<string> staff;      /* шаблон за списък */

   staff.push_back("Cracker, Carl");
   staff.push_back("Hacker, Harry");
   staff.push_back("Lam, Larry");
   staff.push_back("Sandman, Susan");
```

```
   list<string>::iterator pos;     /* итератор на списък */

/* добавя елемент на четвърто място */
   pos = staff.begin();
   pos++;
   pos++;
   pos++;
   staff.insert(pos, "Reindeer, Rudolf");

/* отстранява втория елемент */
   pos = staff.begin();
   pos++;
   staff.erase(pos);

/* добавя елемент на последно място */
   pos = staff.end();
   staff.insert(pos, "Zeider, Zeev");

/* обхождане на списък */
   for (pos = staff.begin(); pos != staff.end(); pos++)
      cout << *pos << "\n";    /* извежда съдържание на текущата позиция */
   return 0;
}
```

```
Cracker, Carl
Lam, Larry
Reindeer, Rudolf
Sandman, Susan
Zeider, Zeev
```

### Стек и опашка

* **Стек** е линейна структура (редица) от данни с добавяне и изтриване (изваждане) на елементи само от единия край, наречен връх на стека.
* Не е позволен достъп до другите елементи на редицата.
* Това правило за добавяне и изваждане на елементи се нарича още LIFO (last in, first out).
* Основни операции:
- добавяне на елемент към стека - `push;`
- и изваждане на елемент от стека - `pop`.
* Двете основни операции отнемат константно време - $O(1)$.
* Реализация в STL.

*Пример:*

```
stack<string> s;
s.push("Tom");
s.push("Dick");
s.push("Harry");
while (s.size() > 0)
{  cout << s.top() << "\n";
   s.pop()
}
```

* **Опашка** е линейна структура (редица) от данни с добавяне на елементи от единия край и изтриване на елементи от другия край.
* Не е позволен достъп до другите елементи на редицата.
* Това правило за добавяне и изваждане на елементи се нарича FIFO (first in, first out).
* Основни операции:
- добавяне на елемент към опашката - `push`;
- изваждане на елемент от опашката - `pop`.
* Двете основни операции се изпълняват за константно време - $O(1)$.
Реализация в STL:
*Пример:*

```
queue<string> q;
q.push("Tom");
q.push("Dick");
q.push("Harry");
while (q.size() > 0)
{  cout << q.front() << "\n";
   q.pop();
}
```

*Пример:*

```
// fifolifo.cpp
01: #include <iostream>
02: #include <string>
03: #include <queue>
04: #include <stack>
06: using namespace std;
07:
08: int main()
09: {  cout << "FIFO order:\n";
11:
12:    queue<string> q;
13:    q.push("Tom");
```

```
14:     q.push("Dick");
15:     q.push("Harry");
16:
17:     stack<string> s;
18:     while (q.size() > 0)
19:     {  string name = q.front();
21:        q.pop();
22:        cout << name << "\n";
23:        s.push(name);
24:     }
26:     cout << "LIFO order:\n";
28:     while (s.size() > 0)
29:     {  cout << s.top() << "\n";
31:        s.pop();
32:     }
34:     return 0;
35: }
```

---

### Други стандартни контейнери

* **Множество** (`set`) е контейнер (линейна структура от данни), който поддържа линейна наредба на елементите (сортирани), независимо в какъв ред са въведени, като не допуска дублиращи се стойности.
* Достъп до елементите на множеството се осъществява посредством итератор.
* Елементите на `set` се съхраняват в нелинейна структура от данни.
* Основните операции `insert` и `erase` имат сложност $O(\log n)$.
* Мултимножество (`multiset`) е подобно на множество, но допуска повече от един елемент с една и съща стойност.

*Пример:*

```
// set.cpp
#include <iostream>
#include <string>
#include <set>
using namespace std;

int main()
{ set<string> s;
  s.insert("Tom");
  s.insert("Dick");
  s.insert("Tom"); // !!!
  s.insert("Harry");

  cout << "set: " << endl;
  set<string>::iterator p;
  for (p = s.begin(); p!= s.end(); p++)
     cout << *p << "\n";
```

```
  multiset<string> ms;
  ms.insert("Tom");
  ms.insert("Dick");
  ms.insert("Tom"); // !!!
  ms.insert("Harry");

  cout << "multiset: " << endl;
  multiset<string>::iterator mp;
  for (mp = ms.begin(); mp!= ms.end(); mp++)
     cout << *mp << "\n";
  cout << ms.size() << endl;
  cout << ms.count("Tom");

  return 0;
}
```

* Контейнер **map** съхранява двойки елементи (ключ, стойност).
* Осигурява ефективен (пряк) достъп до елементите посредством операция индекс.

```
map<string, double> scores;
scores["Tom"] = 90;
scores["Dick"] = 86;
scores["Harry"] = 100;
```

* Multimap е контейнер от двойки елементи (ключ, стойност), като се допускат елементи с еднакви ключове.
* Обект от шаблонът `pair` се състои от два публични елемента (данни) - `first` и `second`, чиито типове се задават като параметри на шаблона.

```
multimap<string, double>
mmap;
mmap.insert(pair("Tom", 90));
mmap.insert(pair("Dick", 86));

mmap.insert(pair("Harry", 100));
mmap.insert(pair("Tom", 190));
```

*Пример:*

```
// map.cpp
#include <iostream>
#include <string>
#include <map>
using namespace std;
```

```
int main()
{ map<string, int> scores;
  scores["Tom"] = 90;
  scores["Dick"] = 86;
  scores["Harry"] = 100;

  map<string, int>::iterator p;
  for (p = scores.begin(); p!= scores.end(); p++)
     cout << p->first << " " << p->second << "\n";
  cout << scores.size() << endl;

  multimap<string, int> mmap;
  mmap.insert(pair<string, int>("Tom", 90));
  mmap.insert(pair<string, int>("Dick", 86));
  mmap.insert(pair<string, int>("Harry", 100));
  mmap.insert(pair<string, int>("Tom", 190));

  multimap<string, int>::iterator q;
  for (q = mmap.begin(); q!= mmap.end(); q++)
     cout << q->first << " " << q->second << "\n";
  cout << mmap.size() << endl;

  return 0;
}
```

---

### Алгоритми в STL

* Причината за въвеждане и използване на итератори в STL е отделянето на алгоритмите от структурите от данни (контейнерите).
* Алгоритмите в STL са реализирани от функции, които работят само с итератори и по такъв начин са независими от контейнерите, където се съхраняват данните.
* *Пример:* Функция, която намира сумата на елементите на контейнер - вектор или списък.

```
vector<double> data;
/* do something with data */

double vsum = 0;
accumulate(data.begin(), data.end(), vsum);
/* now vsum contains the sum of the elements in the vector */

list<double> salaries;
/* do something with salaries */

double lsum = 0;
```

```
accumulate(salaries.begin(), salaries.end(), lsum);
/* now lsum contains the sum of the elements in the list */
```

* Библиотеката поддържа функции за търсене.

```
/* search for a certain name on the staff */
   list<string>::iterator it =
   find(staff.begin(), staff.end(), name);
   cout << *it << endl;
```

* Библиотеката поддържа и функции за сортиране: sort, qsort

---