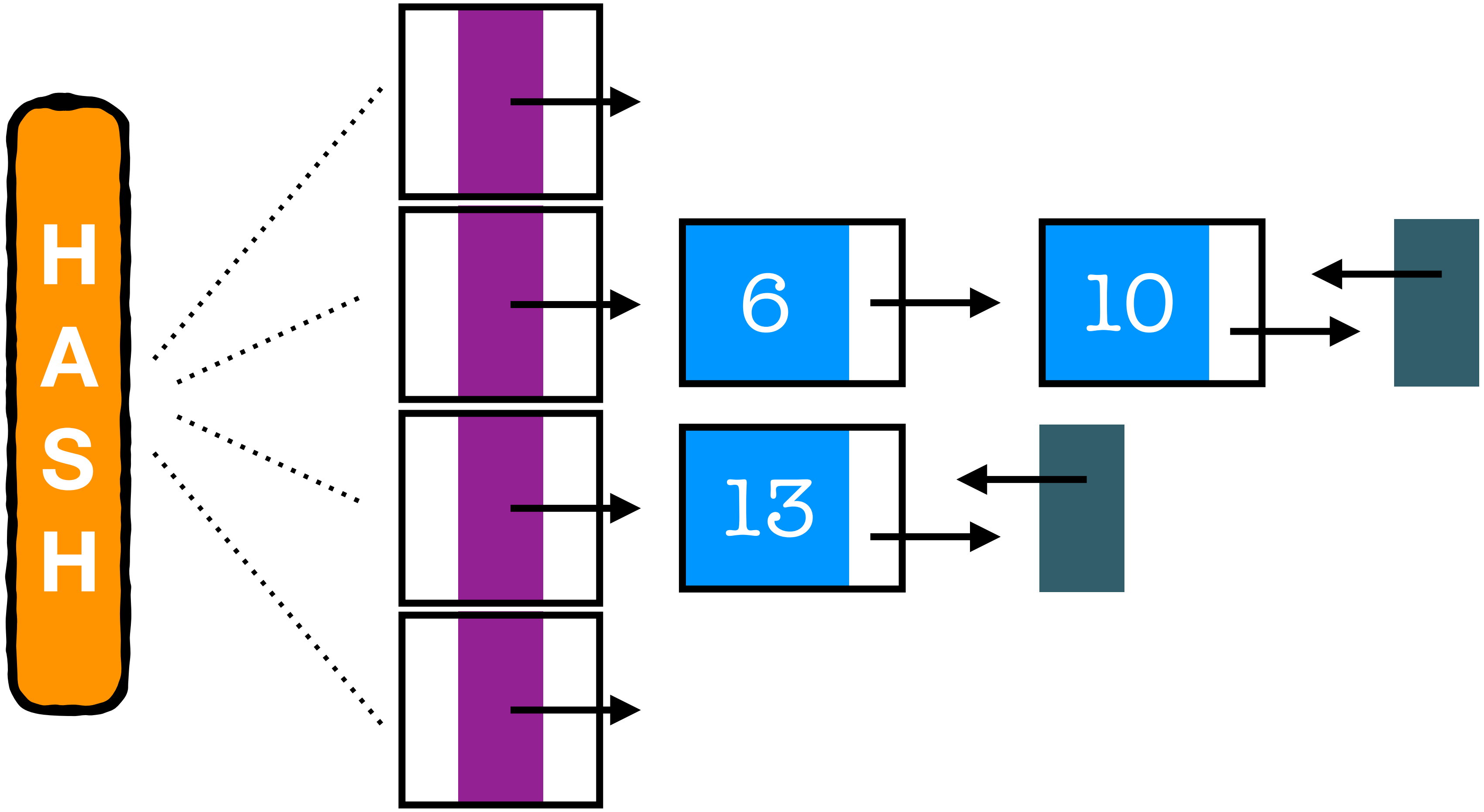
Tabela de
Dispersão

Encadeamento
Simples



chainedHashTable.h

```
template <typename T>
class ChainedHashTable {
  private:
    vector< list<T> > buckets;
    unsigned _size;
  public:
    ChainedHashTable(unsigned size = 5) :
        buckets(size), _size(0) {}

    unsigned size() { return _size; }
    bool empty () { return !size(); }
    unsigned capacity() {
        return buckets.size();
    }
    unsigned load_factor() {
        return (float) _size() / capacity();
    }
```

```
unsigned hash(const T & value) {
    return value >> 1;
}
void print () {
  for (const auto & bucket: buckets) {
    cout << "[] ";
    for (const auto & value: bucket)
      cout << " " << value;
    cout << endl;
}
bool find (const T & value) {
  unsigned idx =
        hash(value) % capacity();
  const auto & pos =
     std::find(buckets[idx].begin(),
               buckets[idx].end(), value);
  return pos != buckets[idx].end();
}
```

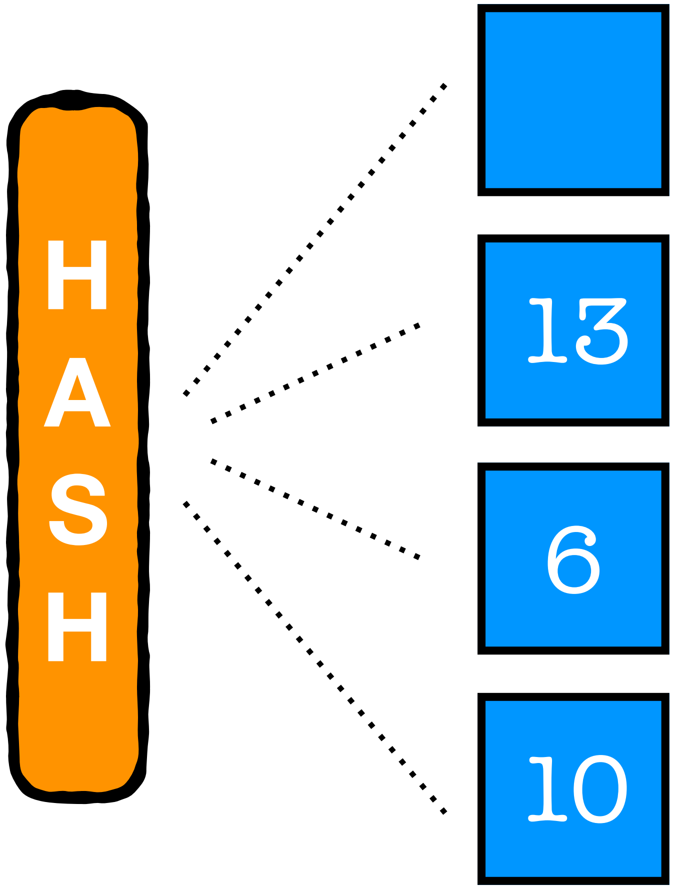
chainedHashTable.h

```
void add (const T & value,
             bool _resize = true) {
   unsigned idx = hash(value) % capacity();
   buckets[idx].push_back(value);
   if (_resize) {
      _size++;
      resize();
   }
}
void del (const T & value) {
   unsigned idx = hash(value) % capacity();
   const auto & pos =
      std::find(buckets[idx].begin(),
                buckets[idx].end(), value);
   if (pos != buckets[idx].end()) {
      buckets[idx].erase(pos);
      _size—; resize();
   }
}
```

```
void resize () {
   unsigned new_size = _size;
   if (load_factor() > 0.7)
      new_size = capacity() * 2;
   else if (load_factor() < 0.2)
      new_size = capacity / 2;
   if (new_size == _size) return;

   vector<T> data;
   data.reserve(_size);
   for (auto & bucket: buckets) {
     for (const auto & value: bucket)
        data.push_back(value);
     bucket.clear();
   }
   buckets.resize(new_size);
   for (const auto & value: data)
      add(value, false);
}
```

Tabela de
Dispersão



Endereçamento
Aberto

chainedHashTable.h

```
template <typename T>
class OpenHashTable {
  private:
    vector<T> container;
    vector<bool> empty, deleted;
    unsigned _size;
    unsigned probing(unsigned start,
                     unsigned attempt) {
      return (unsigned) (start +
             pow(attempt, _probing))
             % capacity();
    }
  public:
    typedef enum
      { LINEAR = 1,
        QUADRATIC = 2 } probing_options;
  private:
    probing_options _probing;
```

```
public:
   OpenHashTable(unsigned size = 5,
      probing_options probing = LINEAR)
      : container(size), empty(size, true),
      deleted(size, false), _size(0),
      _probing(probing) {}

  unsigned size() { return _size; }
  unsigned capacity() {
     return container.size();
  }
  float load_factor() {
     return (float) _size() / capacity();
  }
 unsigned hash(const T & value) {
     return value >> 1;
 }
```

openHashTable.h

```
void print () {
   for (unsigned i = 0; i < capacity(); i++)
      cout << (empty[i] ? “-1” : to_string(container[i])) << " ";
   cout << endl;
}

typename vector<T>::iterator get (const T & value) {
   unsigned idx = hash(value) % capacity();
   unsigned start = idx, attempt = 0;
   do {
      idx = probing(start, attempt++);
      if (!empty[idx] && container[idx] == value) return container.begin() + idx;
   }
   while ((!empty[idx] || deleted[idx])
         && (idx != start || attempt == 1) && attempt < capacity());
   return container.end();
}
```

openHashTable.h

```
bool find (const T & value) {
    return get(value) != container.end();
}

void del (const T & value) {
    auto pos = get(value);
    if (pos == container.end()) return;
    unsigned idx = pos - container.begin();
    empty[idx] = true;
    deleted[idx] = true;
    _size--;
    resize();
}
```

```
void add (const T & value,
                bool _resize = true) {
    unsigned idx = hash(value) % capacity();
    unsigned start = idx, attempt = 0;
    do {
        idx = probing(start, attempt++);
        if (empty[idx]) {
            container[idx] = value;
            empty[idx] = deleted[idx] = false;
            if (_resize) { _size++; resize();}
            return;
        }
    }
    while (!empty[idx]
            && (idx != start || attempt == 1)
            && attempt < capacity());
}
```

openHashTable.h

```
    void resize () {
        unsigned new_size = _size;
        if (load_factor() > 0.7) new_size = capacity() * 2;
        else if (load_factor() < 0.2) new_size = capacity / 2;
        if (new_size == _size) return;

        vector<T> data;
        data.reserve(new_size);
        for (unsigned i = 0; i < capacity(); i++)
          if (!empty[i]) data.push_back(container[i]);

        container.resize(new_size);
        empty.resize(new_size);
        deleted.resize(new_size);
        fill_n(empty.begin(), new_size, true);
        fill_n(deleted.begin(), new_size, false);
        for (auto & value: data)
           add(value, false);
}
```

Operação	TAD Conjunto / Dicionário			
	Tabela de dispersão			
	Encadeamento simples		Endereçamento aberto	
	Melhor	Pior	Melhor	Pior
adicionar	Θ(1)	Θ(n)	Θ(1)	Θ(n)
remover	Θ(1)	Θ(n)	Θ(1)	Θ(n)
pertinência	Θ(1)	Θ(n)	Θ(1)	Θ(n)